# أَتقِن لُغَةَ كُوتلِن

ستيفن ساموِيل
ستيفان بوكيتيو

## دليلك لتعلم لغة كوتلن واحترافها



ترجمة: هشام رزق الله
تحرير: جميل بيلوني
الإعداد والإشراف: فهد بن عامر السعيدي
وادي التقنية

# أَتقِن لُغَةَ كُوتلِن

## دليلك لتعلم لغة كوتلن واحترافها


**ترجمة**

هشام رزق الله

**تحرير**

جميل بيلوني

**الإعداد والإشراف**

فهد بن عامر السعيدي



**الإصدار 1.0**




# مقدمة:

وادي التقنيـة موقـعٌ تقـنيٌ عـربيّ يُعـنى بتتبـع أخبـار البرمجيـات الحـرة والمـواد التعليميـة المتعلقة بها، يكتب فيه عدد من المتطوعين المهتمين بالبرمجيات الحرة والتقنية بشكل عام.

تـأتي ترجمـة هـذا الكتـاب ونشـره في إطـار سـعينا إلى توسـعة قاعـدة الوصـول للتقنيـات الحديثة إلى أكبر شريحة ممكنة من الناطقين باللغة العربية، فبالرغم من وجود الكثـير من الكتب والدورات باللغات الأجنبية إلا أن وجود مصادر عربية أمر لا بد منه، وهذا ما نقوم به بالضبط.

بعد أن نشرنا ترجمـة كتـاب تعلم لغـة GO بسـهولة وكتـاب بوستجريسـكل كتـاب الوصـفات حصـلنا على بعض التبرعـات الماليـة الـتي أتـاح لنـا ترجمـة وإخـراج هـذا الكتـاب، و هـذا الكتـاب مـترجمٌ عن كتـــاب «Programming Kotlin» لمؤلفيـــه ســـتيفن ســـامويل (Stephen Samuel) وسـتيفان بوكيـتيو (Stefan Bocutiu)، والـذي نَشَـرتَه دار نشـر Packt. إن الترجمة العربية هذه مرخصّة بموجب رخصة المشاع الإبداعي «نَسب المُصنَّف 4.0»

دعم وادي التقنيـة تـأليف وترجمـة العديـد من الكتب التقنيـة في مجـال البرمجيـات الحـرة ومفتوحة المصدر، وتوفيرها مجانا للمسـتخدم التقـني العـربي، من أهم الكتب الـتي دعمهـا وادي التقنية: تعلم جافا سكربت، دفتر مدير دبيان، سـطر أوامـر لينُكس، انطلـق في انكسـكيب، تعـرف على البرمجيـات الحـرة، تعلم لغـة GO بسـهولة، كتـاب الشـفرة الكاملـة و بوستجريسـكل كتـاب الوصفات، وغيرها الكثير من الكتب التقنية المتخصصة.

وفي الختام ندعو كل من لديه مشروع لتأليف كتـاب أو قـدرة على الترجمـة الجيـدة باللغـة العربية أن يتواصـل معنـا، لعـل الله أن يوفقنـا على إخـراج المزيـد من الكتب المفيـدة في مجـال تقنية المعلومات.

نسأل الله القبول و التوفيق.

فهد بن عامر السعيدي

مسقط - سلطنة عمان

الأربعاء: 7 شعبان 1441 هـ

1 أبريل 2020م

# جدول المحتويات

# تقديم المحرر



لا يخفى على أي مبرمج ومطور تطبيقات سطوع نجم لغة البرمجة كوتلن وذيـوع صـيتها، إذ هي -لن لا يعرفهـا بعد- لغة حديثة عهد أصدرت عام 2011 واعتمدتها غوغل لغةً رسميةً لتطوير تطبيقات آندرويد منذ عام 2017 بعد أن قرَّرت دعمها لتزاحم جافا في استعمالها آنذاك وتسبقها بكثير من مزايا وتحسينات، وبذلك يصبح لمنصـة آندرويـد لغة كما تعد لغة Swift لغةً رسميةً لتطوير تطبيقات iOS. إن كنت مطور تطبيقـات آندرويـد، فلابـد أن تبـدأ مـع هـذه اللغة بداية قوية لدخول مجال تطوير البرمجيات بقوة. لذا أتى هذا الكتـاب ليشـرح لغـة كـوتلن بـدءًا من الأساسـيات وحتى المفاهيم المتقدمة شرحًا عمليًا مدعَّمًا بالأمثلة القابلة للتطبيق، والتي توضِّح المفاهيم البرمجية الـتي يحـاول هذا الكتاب إيصالها.

ذُكِر في بداية الكتاب أنَّه مقدمة للغة كوتلن ويشرح أساسيات البدء بها دون شرط امتلاكك خبرة مسبقة باللغـة أو أي لغة برمجية أخرى مثل جافا أو سكالا، إلا أنَّني وجدت بعد مراجعة الكتاب أنَّ مستوى الكتـاب من متوسـط إلى متقدم وعليك امتلاك خبرة مسبقة -بسيطة على الأقل- بكوتلن أو أي لغة برمجيـة أخـرى لتحقيـق أكـبر اسـتفادة من هذا الكتاب.

أنصح أثناء البدء بقراءة الكتاب بوضع توثيق كوتلن في موسوعة حسوب تحت متناول يدك دومًا وأن تقـرأ كـل قسم التوثيق المقابل للموضوع الذي يتحدث عنه كل فصل، ولتسهيل ذلك، حاولت جاهدًا توفير روابط في بداية كل فصل للانتقال إلى القسم المقابل في التوثيق. الجدير بالذكر أن توثيق كوتلن ذاك قد راجعته مراجعةً دقيقةً وحـرَّرت كامل صفحاته وحسَّنتها وزدت عليها عن التوثيق الأجنبي حتى أضحت أفضـل من توثيـق اللغـة الرسـمي نفسـه، ولن تجد اختلافًا في الأسلوب أو المصطلحات بين هذا الكتاب والتوثيـق العـربي ذاك، وبـذكر المصـطلحات، فقـد وضـعت المصطلحات الأجنبية بجانب العربية لفك صلاسم الترجمـة العربيـة الـتي قـد تسـمع بهـا لأول مـرة خصوصًـا إن كنت معتادًا على القراءة والتعلم من المصادر الأجنبية. حاولت أيضًا توفير روابـط في ويكيبيـديا وتوثيـق كـوتلن الرسـمي وغيرهما من مصادر أجنبية لأي مواضيع رأيت أنها بحاجة إلى مصدر خارجي آخر للاستزادة والترسـيخ راجيًـا بـذلك تحقيق أكبر فائدة.

أضفت أيضًا على الهوامش ملاحظات وجدت أنَّها ستفيدك وتعينك أثناء رحلتك هـذه، أخص منهـا المصـطلحات العربية المترجمة المقابلة لمصطلحات برمجية أجنبية (لم يسبق أن تُـرجَمت من قبـل) وإضـافات شـرحية وتنبيهـات حول انتهاء صلاحية الإصدار المستعمل للغة أو المكتبة وغيرها.

أرجو أن تنتبه إلى جملتين: الأولى «أثناء ترجمة هذا الكتاب» والثانية «أثناء مراجعة أو تدقيق هـذا الكتـاب» إذ عملت على تحديث إصدار كوتلن وإصدارات المكتبات وأطر العمل وما يقابلها من معلومـات وشـيفرات مـا اسـتطعت إلى ذلك سبيلًا وبقي جزء آخر غير مُحدَّث سنضيفه في إصدارات لاحقـة، لـذا تشـير الجملـة الأولى إلى الـوقت الـذي كُتِب فيـه الكتـاب، عـام 2017، والثانيـة إلى وقت مراجعـة الكتـاب وتدقيقـه، أواخـر عـام 2019، فـإن واجهت أي مشكلة، فتأكد من موضـوع الإصـدارات وارجـع دومًـا إلى سـجل التغيـيرات Changelog في التوثيقـات والمواقـع الرسمية.

وفي النهاية، أحمد الله على توفيقه بإتمام العمل على الكتاب، وأرجو أن يكـون إضـافةً مفيـدةً للمكتبـة العربيـة، والله ولي التوفيق.

جميل عبد الله بيلوني

إسطنبول، تركيا

31/3/2020

# تمهید



ترتبط كوتلن في العادة بتطوير تطبيقات أندرويد، وتدور معظم النقاشات حول ذلك، لكن لدى هذه اللغة الكثير لتقدمه وهي مثاليَّة لمطوري جانب الخادم في الوقت الحالي، على الرغم من أن أي مطور أندرويد سيجد مقتطفات وأمثلة مفيدة في هذا الكتاب، إلا أن الكتاب يستهدف مطوري جافا (Java) وسكالا (Scala) بشكل أساسي. سيبدأ هذا الكتاب بمقدمة للغة كوتلن وسيشرح كيفية إعداد البيئة قبل الانتقال إلى المفاهيم الأساسية، وبمجرد الانتهاء من الأساسيات، سيركز على مفاهيم أكثر تقدمًا، ولا تتفاجأ إذا رأيت بعض شيفرات البايتكود، وفي النهاية عند إنهاء الكتاب، ستكون مستعدًا لاستخدام كوتلن في مشروعك القادم.

## 1. ما يغطيه هذا الكتاب

الفصل الأول، البدء مع كوتلن، يشرح كيفية تثبيت كوتلن، Jetbrains Intellij IDEA ونظام البناء Gradle، وبمجرد الانتهاء من تثبيت الأدوات، سيوضح لك الفصل كيفية كتابة برنامج كوتلن الأول.

الفصل الثاني، أساسيات كوتلن، ستبدأ بالغوص في أساسيات كوتلن، بما في ذلك، الأنواع الأساسيَّة، الصياغة الأساسيَّة، وهياكل تدفق التحكم مثل تعليمات `if` الشرطيَّة (if statements) وحلقات `for` و `while`، ويُختتم الفصل بإضافات محددة للكوتلن مثل تعليمات `when` واستدلالات النوع.

الفصل الثالث، البرمجة كائنية التوجه في كوتلن، التركيز على جانب الكائن المتوجه في اللغة، وسنتحدَّث عن الأصناف، الواجهات، الكائنات و العلاقات بينهما، الأنواع الفرعيَّة والتعددية الشكليَّة (polymorphism).

الفصل الرابع، الدوال في كوتلن، يتحدث عن الدوال (وتُعْرف أيضًا بالإجراءات أو التوابع) التي هي كتل البناء الأساسيَّة لأي لغة، وسيغطي هذا الفصل الصياغة، بما في ذلك تحسينات كوتلن مثل المعاملات المسماة، والمعاملات الافتراضية، والدالة المُجرَّدة.

الفصل الخامس، الدوال الأعلى مرتبةً والبرمجة الوظيفيَّة، يركز على جانب البرمجة الوظيفية في كوتلن، بما في ذلك المغلفات (والتي تعرف أيضا بلامدا lambda) ومراجع الدالة، كما يغطي أيضا تقنيات البرمجة الوظيفية مثل التطبيق الجزئي، وتركيب الدالة وتراكم الخطأ.

الفصل السادس، الخاصيات، يشرح لك أن الخاصيات (properties) تعمل جنبًا إلى جنب مع برمجة الكائن الموجَّه للكشف عن القيم في كائن أو صنف، ويغطي هذا الفصل كيفية عمل هذه الخاصيات، وكيف يمكن للمستخدم

تحقيق أقصى استفادة منها، وكذلك كيفية تولّد في البايتكود (bytecode).

الفصل السابع، أمان القيم الفارغة، والانعكاس، والتوصيفات، تشرح أن سلامة العَدم Null هي واحد من أهم المميزات التي يوفرها كوتلن، ويغطي الجزء الأول من هذا الفصل الأسباب والكيفيات حول سلامة العَدم null في كوتلن، ويقدم الجزء الثاني من هذا الفصل الانعكاس (مراقبة الشيفرة البرمجية وقت التشغيل)، وكيف يمكن استخدامها في البرمجة الوصفيّة (meta programming) مع التوصيفات.

الفصل الثامن، التعميم، والأنواع المُعمَّمَة، يشرح هذا الفصل الأنواع مُعمَّمَة (Generics) أو أنواع المعاملات والتي هي عنصر أساسي من أي نظام نوع متقدم، ونظام النوع في كوتلن متطور بشكل كبير عن ذلك الموجود في جافا، ويغطي هذا الفصل التباين بما في ذلك نوع اللاشيء Nothing وأنواع البيانات الجبرية.

الفصل التاسع، أصناف البيانات، يبيِّن لك أن الأصناف الثابتة ومجال الأصناف الحرّة هو موضوع ساخن في الوقت الحالي وذلك بسبب الطريقة التي تسهّل بها الشيفرة البرمجية وتبسط مزامنة البرمجة، وتمتلك كوتلن العديد من المميزات التي تركز على هذا المجال، والتي تسمى أصناف البيانات.

الفصل العاشر، التجميعات، يشرح لك أن التجميعات هي واحدة من أكثر الجوانب استخدامًا من أي مكتبة قياسيّة، وتجميعات جافا ليست استثناء. يصف هذا الفصل التحسينات التي قدمتها كوتلن إلى تجميعات JDK، بما في ذلك العمليات الوظيفيَّة مثل `map` و `fold` و `filter`.

الفصل الحادي عشر، الاختبار في كوتلن، يفسّر لك أن استخدام اللغة لتجربة الشيفرات البرمجيَّة هي واحدة من البوابات إلى أي لغة جديدة، ويوضح لك هذا الفصل كيف يمكن أن يُستخدَم إطار التجربة KotlinTest لكتابة التعابير والاختبارات القابلة للقراءة بشكل أقوى من التي تسمح بها اختبارات jUnit القياسيَّة.

الفصل الثاني عشر، الخدمات المصغَّرة في كوتلن، يبيِّن لك كيف أصبحت الخدمات المصغَّرة (Microservices) تهيمن على معماريّة جانب الخادم في السنوات الأخيرة، وكوتلن هو خيار ممتاز لكتابة مثل هذه الخدمات، سيقدم لك هذا الفصل إطار Lagom microservice وسيشرح كيفيَّة استخدامه لعمل تأثيرات كبيرة مع كوتلن.

الفصل الثالث عشر، التزامن، سيشرح لك أنَّ البرامج متعدد النوى (multi-core) تصبح أكثر أهمية في إطارات جانب العميل، وسيركز هذا الفصل على المقدمة القوية لتقنيات البرمجة المتزامنة المستخدمة في التطوير

الحديث، بما في ذلك، الخيوط، والأنواع الأولية المتزامنة بنى المستقبل (futures).

## 2. ما الذي تحتاج إليه مع هذا الكتاب؟

تحتـاج في هـذا الكتـاب إلى حاسـوب يعمـل على نظـام مـاك، أو لينكس أو ويـنـدوز قـادر على تشـغيل أحـدث إصـدارات جافـا، ومن المستحسـن أن يمتلـك الجهـاز ذاكـرة قـادرة على تشـغيل النسـخة الحديثـة من بيئـة التطـوير المتكاملة Jetbrains' Intellij IDEA.

## 3. لمن هذا الكتاب؟

هذا الكتاب للأشخاص الذين يمتلكون خبرة قليلة أو معدومة في لغـة كـوتلن ويرغبـون في تعلم اللغـة بسـرعة. يركز هذا الكتاب على التطوير في جانب الخادم في كوتلن وسيكون مناسـبًا لمطـور من طـرف الخـادم أو من يـرغب في تعلم ذلـك ولا تلـزم معرفـة مسـبقة بالبرمجـة الوظيفيـة أو البرمجـة كائنيَّة التوجـه، لكن ينصـح ببعض المعرفـة باللغات الأخرى.

تحتوي بعض الفصول على أقسام موجزة توازن بين تطبيقات جافا مع ابنة عمتهـا كـوتلن ويمكنـك تخطي هـذه الصفحات بالنسبة للأشخاص الذين لا يمتلكون معرفة مسبقة بجافا.

## 4. تحميل الشيفرة البرمجية للأمثلة

يمكنك تحميل ملفات الشيفرة البرمجية لهذا الكتاب من صفحة الكتاب على GitHub. أو يمكنـك زيـادة موقـع packtpub.com لتحميل ملفات الشيفرات البرمجية عن طريق اتباع الخطوات التالية:

1. سجّل دخولك باستخدام بريدك الإلكتروني وكلمة المرور.
2. مرِّر مؤشِّر الفأرة على صفحة SUPPORT في الأعلى.
3. اضغط على Code Downloads & Errata.
4. اكتب اسم الكتاب الأجنبي في مربع البحث، Programming Kotlin.
5. حدّد الكتاب الذي تبحث عنه لتنزيل الملفات.
6. اختر من القائمة المنسدلة مكان الذي اشتريت الكتاب منه.

7. اضغط على Code Download.

يمكنك أيضا تحميل ملفات الشيفرة البرمجية عن طريق الضغط على زر Code Files في صفحة الكتاب في موقع Packt Publishing، ويمكنك الوصول إلى هذه الصفحة عن طريق كتابة اسم الكتاب في مربع البحث، ولاحظ أنك تحتاج إلى تسجيل الدخول إلى حساب Packt.

بمجرد تحميل الملف، يرجى التأكد من فك ضغط واستخراج المجلد باستخدام أحدث إصدار من:

- WinRAR أو 7-Zip لنظام ويندوز.
- Zipeg أو iZip أو UnRarX لنظام ماك.
- 7-Zip أو PeaZip لنظام لينكس.

## 5. أخطاء مطبعيَّة

رغم أنَّنا كنا حذرين لضمان دقة المحتوى إلا أن الأخطاء تحدث، لذلك، إذا وجدت خطأ في أحد كتبنا، ربما خطأ في النص أو الشيفرة البرمجية، سنكون ممتنين إذا كان بإمكانك إبلاغنا، حتى تنقذ القراء الآخرين من هذه الأخطاء وتساعدنا على تحسن الإصدارات اللاحقة من هذا الكتاب، فإذا وجدّت أية أخطاء، يرجى إبلاغنا عنها من خلال زيارة itwadi.com/contact ثم كتاب اسم الكتاب في مربع الموضوع وإدخال المعلومات الخطأ، وبمجرد التحقق من هذا الخطأ، سيُقبَل إرسالك وسيصحح الخطأ في الإصدار التالي من النسخة المترجمة.

الفصل الأول:

# البدء مع كوتلن



لقد حان الوقت لكتابة التعليمات البرمجية! سنبدأ في هذا الفصل بكتابة الشيفرة البرمجيَّة الأولى التي يُبدَأ بها عند تعلم أية لغة برمجية ألا وهي عبارة «أهلا بالعالم!» (Hello World!) الشهيرة. وللقيام بذلك، سنحتاج أولًا إلى تهيئة البيئة اللازمة لتطوير البرمجيات مع كوتلن، وسنقدم بعض الأمثلة باستخدام تشغيل المُصرِّف (Compiler) من سطر الأوامر، ومن ثم سننتقل إلى الطريقة النموذجية للبرمجة وذلك باستخدام بيئة تطوير متكاملة (IDE اختصارٌ للعبارة Integration Development Environment) وأية أدوات بناء متاحة.

كوتلن هي لغةٌ مستندةٌ إلى آلة جافا الافتراضية (JVM، اختصارٌ للعبارة Java Virtual Machine)، وسيولد بالتالي المصرَّف شيفرةً بلغة تدعى «جافا بايتكود» (Java bytecode، وهي لغة تفهمها آلة جافا الافتراضية وتأخذ عادةً اللاحقة class.)، ويمكن بعدئذٍ لشيفرة كوتلن استدعاء الشيفرات البرمجية لجافا والعكس بالعكس؛ ولذلك، ستحتاج إلى تثبيت أدوات جافا التطويرية (JDK، اختصارٌ للعبارة Java Development Kit) على جهازك إن لم تكن مثبَّتةً من قبل.

ولتتمكن من كتابة شيفرات برمجية للآندرويد، إذ أن أحدث إصدار جافا مدعوم هو 6، سيحتاج المصرِّف إلى تصريف الشيفرة البرمجية إلى لغة بايتكود متوافقة مع الإصدار السادس من جافا؛ وفي هذا الكتاب، ستعمل إن جميع الأمثلة مع الإصدار Java JDK 8، فإذا كنت جديدًا في عالم JVM، يمكنك الحصول على الإصدار الأخير من موقع oracle.com.

ستتعلَّم في هذا الفصل كيف:

- تستخدم سطر الأوامر لتصريف وتنفيذ شيفرة برمجية مكتوبة بلغة كوتلن.
- تستخدم REPL (حلقة اقرأ-قيِّم-اطبع، Read–Eval–Print Loop) وتكتب سكربتات كوتلن.
- تنشئ مشروع gradle مع تفعيل كوتلن.
- تنشئ مشروع Maven مع تفعيل كوتلن.
- تستخدم بيئة IntelliJ التطويرية لإنشاء مشروع كوتلن.
- تستخدم بيئة Eclipse IDE التطويرية لإنشاء مشروع كوتلن.
- تدمج بين شيفرات جافا وكوتلن في نفس المشروع.

## 1. استخدام سطر الأوامر لتصريف وتشغيل شيفرة كوتلن

ستحتاج إلى مُشغِّل آني (runtime) ومُصرِّف (compiler) خاص بكوتلن لكتابة وتنفيذ الشيفرات البرمجية المكتوبة بها. في وقت مراجعة هذا الكتاب، ستجد الإصدار 1.3 متاحًا (الإصدار المستقر هو 1.3.31، وقد يكون هنالك إصدار أحدث منه أثناء قراءة هذا الكتاب)، ويأتي مع كل إصدار مُشغِّل آني (runtime) جديد نسخةً خاصةً من المصرِّف؛ وللحصول عليه، اذهب إلى هذا الرابط، وانتقل إلى أسفل الصفحة، ونزِّل وفك ضغط الملف kotlin-compiler-1.3.31.zip إلى المكان الذي ترغب به على جهازك. وسيحتوي الملف النهائي على المجلد الفرعي bin الذي يحوي جميع السكربتات المطلوبة لتصريف وتشغيل كوتلن على ويندوز، أو لينكس أو ماك. وستحتاج بعدئذٍ للتأكد من أن الملف bin هو جزءٌ من متغير البيئة PATH في نظامك حتى تتمكن من استدعاء kotlinc دون الحاجة إلى تحديد المسار الكامل.

إذا كنت تستخدم لينكس أو ماك، فهنالك طريقة أسهل لتثبيت المصرِّف باستخدام sdkman، إذ كل ما تحتاج له هو تشغيل الأوامر التالية في الطرفيَّة:

```
$ curl -s https://get.sdkman.io | bash
$ bash
$ sdk install kotlin 1.3.31
```

بدلًا من ذلك، إذا كنت تستخدم ماك وكان homebrew مثبّتًا على جهازك، يمكنك تشغيل الأوامر التالية لإنجاز نفس الشيء:

```
$ brew update
$ brew install kotlin@1.3.31
```

والآن بعد الانتهاء من كل هذا، يمكننا كتابة شيفرة كوتلن الأولى. سيعرض التطبيق الذي سنكتبه العبارة «أهلًا بالعالم!» على الطرفية، ابدأ بإنشاء ملف جديد باسم HelloWorld.kt واكتب التالي فيه:

```
fun main(args: Array<String>) {
     println("أهلا بالعالم!")
}
```

استدعِ المصرِّف من سطر الأوامر لإنتاج شيفرة بصيغة JAR التجميعيَّة (include-runtime هي راية تحت

المصرّف على إنتاج ملف JAR قائمًا بذاته وقابل للتشغيل عن طريق تضمين مُشغِّل كوتلن الآني إلى الشيفرة التجميعيّة الناتجة).

```
$ kotlinc HelloWorld.kt -include-runtime -d HelloWorld.jar
```

يمكنك الآن تشغيل البرنامج عن طريق كتابة التالي في سطر الأوامر، إذ يُفترَض أنك ضبطت المتغير `JAVA_HOME` وأضفته إلى مسار النظام:

```
$ java -jar HelloWorld.jar
```

بعد تنفيذ هذا السطر، يجب أن ترى العبارة «أهلا بالعالم!» طُبعَت في الطرفية.

الشيفرة البرمجية واضحة للغاية، فهي تُعرِّف دالةً تُعدُّ نقطة الدخول للبرنامج، إذ تحوي تلك الدالة أمرًا وحيدًا وهو طباعة نص على الطرفيَّة.

إذا كنت تستخدم لغة جافا أو سكالا (Scala)، فستلاحظ باستغراب اختفاء الكلمة المفتاحية `class` التي تمثِّل الصنف الذي سيُعرِّف نقطة دخول البرنامج الرئيسي `main` المتصف عادةً بكونه static.فكيف يعمل هذا إذًا؟ لنلقِ نظرة على ما يحدث فعلًا. دعنا نصرِّف التعليمات البرمجية السابقة بتشغيل الأمر التالي الذي سينشئ الملف `HelloWorld.class` في نفس المجلَّد:

```
$ kotlinc HelloWorld.kt
```

والآن، بعد توليد ملف بلغة جافا بايتكود، يمكننا إلقاء نظر باستخدام الأداة `javap` الموجودة مع أدوات جافا التطويريةJDK (تأكد من أن الملف يحتوي على اللاحقة `kt`):

```
$ javap -c HelloWorldKt.class
```

بمجرد اكتمال التنفيذ، سترى شيئًا شبيهًا بما يلي مطبوعًا في الطرفية:

```
Compiled from "hello.kt"
public final class HelloKt {
  public static final void main(java.lang.String[]);
    Code:
```

```
         0: aload_0
         1: ldc           #9        // String args
         3: invokestatic  #15       // Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/
Object;Ljava/lang/String;)V
         6: ldc           #17    // String !أهلا بالعالم
         8: astore_1
         9: iconst_0
        10: istore_2
        11: getstatic     #23       // Field java/lang/System.out:Ljava/io/
PrintStream;
        14: aload_1
        15: invokevirtual #29       // Method java/io/PrintStream.println:
(Ljava/lang/Object;)V
        18: return
}
```

لا يجب أن تكون خبيرًا بلغة جافا بايتكود لتفهم ما يفعله المصرِّف هنا، إذ أُنشئ - كما ترى في المقتطف السابق - صنفٌ لنا يحتوي على نقطة دخول البرنامج مع تعليمات لطباعة العبارة «أهلا بالعالم!» في الطرفيَّة.

لا أتوقع منك العمل مع مصرِّف سطر الأوامر بشكل يومي؛ وبدلًا من ذلك، يجب عليك استخدام أدوات لتفويض وأتمتة هذه العملية برمتها كما سنرى قريبًا.

## 2. مُشغِّل كوتلن الآني

عندما صرَّفنا الملف HelloWorld وأنتجنا ملفًا باللاحقة JAR، أوعزنا المصرِّف إلى حزمة في مُشغِّل كوتلن الآني (Kotlin runtime)، فلماذا نحتاج إلى مُشغِّل آني؟ ألقِ نظرة فاحصة على الملف الذي جرى توليده بلغة بايتكود إذا لم تكن قد فعلت ذلك بالفعل أو على السطر 3 - لنكون أكثر تحديدًا -الذي يستدعي تابعًا للتحقق من أن المُتغيِّر args ليس فارغًا؛ وبالتالي، إذا صرَّفت الشيفرة البرمجية دون طلب تحزيم مُشغِّل آني معها وحاولت تشغيلها، فستحصل على استثناء (exception). حسنًا، دعنا نجرِّب ذلك:

```
$ kotlinc HelloWorld.kt -d HelloWorld.jar
```

```
$ java -jar HelloWorld.jar
Exception in thread "main" java.lang.NoClassDefFoundError:
kotlin/jvm/internal/Intrinsics at HelloWorldKt.main(HelloWorld.kt)
Caused by: java.lang.ClassNotFoundException:
kotlin.jvm.internal.Intrinsics
```

بصمة المُشغِّل الآني صغيرة للغاية، إذ هي بحجم 800 كيلوبايت تقريبًا ولا جدال حول ذلك، فيأتي كوتلن مع مكتبة صنف قياسيَّة خاصة به (Kotlin runtime) والتي تختلف عن مكتبة JAVA؛ ونتيجةً لذلك، تحتاج إلى دمجها في الملف JAR الناتج، أو توفيرها في مسار الصنف `classpath`:

```
$ java -cp $KOTLIN_HOME/lib/kotlin-runtime.jar:HelloWorld.jar HelloWorldKt
```

إذا طوَّرت مكتبةً للاستخدام الحصري لمكتبات أو تطبيقات كوتلن أخرى، فلن تحتاج عندئذٍ إلى تضمين المُشغِّل الآني، إذ هنالك بدلًا من ذلك طريقة أقصر وهي عن طريق تمرير راية إلى مصرِّف كوتلن بالشكل التالي:

```
$kotlinc -include-runtime HelloWorld.kt -d HelloWorld
```

## 3. الصدفة التفاعلية مع الأداة REPL

توفِّر معظم اللغات هذه الأيام صدفة تفاعليَّة (interactive shell)، وكوتلن واحد من هذه اللغات. فإذا أردت كتابة بعض التعليمات البرمجية بسرعة دون حاجتك لاستخدامها مجددًا، فستكون أداة REPL هي بالضبط ما تحتاجه. يُفضِّل البعض اختبار التوابع الخاصة بهم بسرعة، لكن يجب عليك دائمًا كتابة وحدة اختبارات بدلًا من استخدام REPL للتحقق من صحَّة المخرجات.

يمكنك بدء REPL عن طريق إضافة الاعتماديات إلى مسار الصنف (classpath) وذلك لتكون متاحةً داخل النسخة (instance). سنستخدم مثالًا على ذلك المكتبة Joda للتعامل مع التاريخ والوقت، وسنحتاج أولًا لتحميل الملف JAR.

استعمل، في نافذة الطرفيّة، الأوامر التاليَّة:

```
$ wget
https://github.com/JodaOrg/joda-time/releases/download/v2.10.1/joda-time-
```

```
2.10.1-dist.tar.gz
$ tar xvf joda-time-2.10.1-dist.tar.gz
```

أنت مستعد الآن لاستخدام REPL. أرفق المكتبة Joda إلى نسختها التي هي قيد العمل واستدعِ واستخدم الأصناف التي توفِّرها:

```
$ kotlinc-jvm -cp joda-time-2.10.1/joda-time-2.10.1.jar
Welcome to Kotlin version 1.3.31 (JRE 11.0.2+9-Ubuntu-3ubuntu118.04.3)
Type :help for help, :quit for quit
>>> import org.joda.time.DateTime
>>> DateTime.now()
res1: org.joda.time.DateTime! = 2019-05-03T18:52:10.416+03:00
```

## 4. سكربتات مكتوبة بكوتلن؟!

يمكن تشغيل الشيفرة المكتوبة بكوتلن مثل السكربت. فإذا لم ترغب باستخدام Bash أو Perl، فلديك بديل الآن.

إذا افترضنا أنك تريد حذف جميع الملفات التي مضى عليها أكثر من س يومًا، فإليك السكربت التالي الذي يفعل ذلك:

```
import java.io.File
val purgeTime = System.currentTimeMillis() - args[1].toLong() * 24 * 60 *
60 * 1000
val folders = File(args[0]).listFiles { file -> file.isFile }
folders ?.filter {
file -> file.lastModified() < purgeTime }
?.forEach {
file -> println("Deleting ${file.absolutePath}")
file.delete()
}
```

أنشئ ملفًا باسم `delete.kts` وضع فيه المحتوى السابق، ولاحظ المتغيِّر `args` المعرّف مسبقًا الذي يحتوي على جميع المعاملات المُمرَّرة إلى السكربت عند استدعائه. وقد تتساءل عن الذي يفعله الرمز ? هنالك! فإذا كنت معتادًا على البرمجة باستخدام لغة #C وتعرف الأصناف `nullable`، فستكون على علمٍ مسبقٍ بالجواب؛ وحتى لو

لم تقرأ أو تبحث عنها، فأنا متأكد أن لديك فكرة جيّدة حول عملها، إذ يُدعَى هـذا المحـرف بمعامـل الاسـتدعاء الآمن، وهو - كما ستعرف لاحقًا في هذا الكتاب - أنه يتجنب الخطأ `NullPointerException`.

يأخذ السكربت معاملين: المجلد الهدف المراد حذف ملفاته القديمة وعدد الأيام الذي يمثِّل الحد الفاصل لحذف أو إبقاء الملف. سيتأكد هذا السكربت لكل ملف سيجده في المجلد الهدف من آخر مرة عُدّل فيها وسـيحذفه إذا كـان أقل من الوقت المحسوب بناءً على عدد الأيام المعطى. ولقد تُرِكَ خطـأً بسـيط في السـكربت السـابق يجب معالجتـه قبل تنفيذه، وسنترك هذا للقارئ على أنَّه أول تمرين.

أصبح السكربت مكتملًا الآن، ويمكنك استدعاؤه وتنفيذه عن طريق الأمر التالي:

```
$ kotlinc -script delete.kts . 5
```

إذا نسـخت أو أنشـأت ملفـات في المجلـد الحـالي وكـان آخـر وقت عُـدِّل عليهـا أقـدم من 5 أيـام، فسيتم إزالتها.

## 5. كوتلن مع Gradle

إذا كنت معتادًا على استخدام أدوات البناء، فستكون في أحد هذه التجميعات: Maven، أو Gradle أو SBT (على الأرجح إذا كنت مبرمج Scala). ولن أدخل في التفاصيل، لكننا سنقدم أساسيات Gradle، وهـو نظـام أتمتـة آلي ذو بنيـة متعـددة اللغـات ومفتـوح المصـدر، ويمكنـك معرفـة المزيـد من المعلومـات حولـه من موقعـه الرسـمي gradle.org.

قبل المتابعة، أرجو منك التأكد من تثبيته وتوافره في مسـار الأصـناف (`classpath`) من أجـل الوصـول إليـه من الطرفية. فإذا كان لديك SDKMAN (شرحنا كيفية تثبيته مسبقًا)، فيمكنـك تثـبيت Gradle عن طريقـه وذلـك باستخدام الأمر التالي:

```
$ sdk install gradle 5.4.1
```

يأتي نظام البناء مع بعض القوالب الجـاهزة، وأصبحت كـوتلن مُضـمَّنةً بـدءًا من الإصـدار 5، إذ جـاءت الإضـافة Gradle Kotlin DSL بشكلها التطويري في الإصدار 4 من gradle لتضـيف دعم Kotlin إلى gradle بالشـكل التي يتيح استعمالها عوضًا عن Groovy. دعنا نرى أولا كيف يمكنك معرفة القوالب المتاحة:

```
$ gradle help --task :init
```

يجب أن ترى في قسم الخيار type– شيئًا شبيهًا بما يلي:

```
Options
         --type     Set the type of project to generate.
               Available values are:
                    basic
                    cpp-application
                    cpp-library
                    groovy-application
                    groovy-library
                    java-application
                    java-library
                    kotlin-application
                    kotlin-library
                    pom
                    scala-library
```

قبل إنشاء القالب، يجب أن ننشئ مجلّدًا جديدًا لحواية ملفات المشروع الخاص بنا. استعمل الأمر التالي لإنشاء مجلد جديد باسم HelloWorld:

```
$ mkdir HelloWorld
```

لنذهب ونستعمل قالب كوتلن kotlin-application وننشئ هيكل مشروعنا من خلال تنفيذ الأمر التالي:

```
$ gradle init --type kotlin-application
```

أو يمكنك بدلًا من ذلك كتابة الأمر gradle init فقط الذي سيُظهِر القوالب المتوافرة لاختيار واحدة منها بالشكل التالي:

```
Select type of project to generate:
   1: basic
   2: cpp-application
   3: cpp-library
   4: groovy-application
   5: groovy-library
```

```
   6: java-application
   7: java-library
   8: kotlin-application
   9: kotlin-library
   10: scala-library
Enter selection (default: basic) [1..10]
```

اكتب في هـذه الحالـة الـرقم 8 لتحديـد القـالب `java-application`. سـواءً اسـتعملت هـذه الطريقة أو حدَّدت القالب منذ البداية عبر تمرير اسمه للراية `--type` فإنك ستصل للخطوة التالية وهي تحديد اللغـة التي ستستعلمها مع gradle:

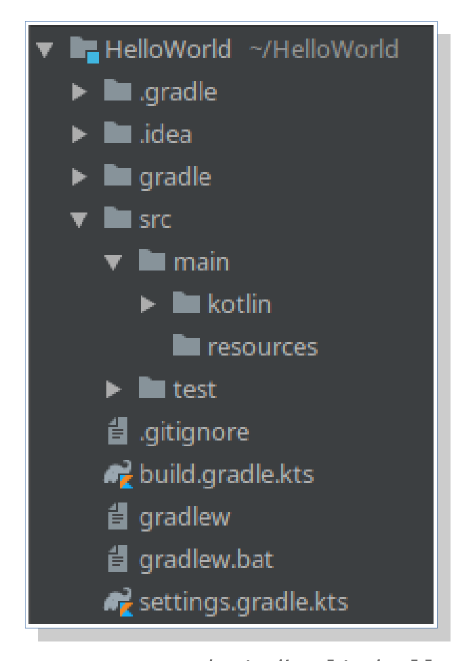
```
Select build script DSL:
 1: groovy
 2: kotlin
Enter selection (default: kotlin) [1..2]
```

كـانت groovy هي اللغـة الشـائع اسـتعمالها قبـل إضـافة Kotlin. والآن، مـع إضـافة Kotlin، يمكنـك تحديدها واستعمالها ولن تحتاج إلى بعد الآن إلى تعلم groovy لكتابة سكربتات gradle بها. اكتب الرقم 2 في الطرفية لاختيار الخيار Kotlin.

ستُسَأل بعد ذلك عن اسم المشروع (Project name) وعن مصدر الحزمة (Source package) واللذان سيُضبطَان إلى اسم المجلَّد `HelloWorld` الذي أنشأنا المشروع فيه افتراضيًا. سنعتمد على التسمية الافتراضية بالضغط على زر الإدخال (enter).

```
Project name (default: HelloWorld):
Source package (default: HelloWorld):
```

سيولِّد هذا القالب مجموعة من الملفات والمجلدات، وسترى هيكلًا مشابهًا للهيكل التالي:




ستلاحظ في المجلد `src/main/kotlin/HelloWorld` وجود ملف يدعى `App.kt` يحوي شيئًا يشبه ما يلي:

```
/*
 * This Kotlin source file was generated by the Gradle 'init' task.
 */
package HelloWorld
class App {
     val greeting: String
          get() {
               return "Hello world."
          }
}
fun main(args: Array<String>) {
     println(App().greeting)
}
```

لا تقلق إن لم تفهم شيئًا منه بعد. هذه الشيفرة تظهر الرسالة الترحيبية الشهيرة التي يكتبها أي مبرمج بدأ طريقه في تعلم البرمجة. حسنًا، الشيفرة مكتوبة وجاهزة للتنفيذ ولكن كيف سنعتمد على gradle من أجل بناء الشيفرة وتنفيذها؟ إن أردت بناء الشيفرة ثم تنفيذها، نفذ الأمرين التاليين على التوالي:

```
$ gradle build
$ gradle run
```

يجب أن تحصل على شيء يشبه الناتج التالي:

```
$ gradle build

BUILD SUCCESSFUL in 10s
8 actionable tasks: 8 executed
$ gradle run

> Task :run
Hello world.

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
```

تخبرنا الرسالة الأولى أن عملية بناء (تصريف) الشيفرة تمت بنجاح، والثانية هي أن تنفيذ الشيفرة جرى بنجاح أيضًا وأظهر الرسالة "Hello world" التي ناتج تنفيذ الدالة `println(App().greeting)` (وظيفية هذه الدالة هي طباعة شيء على الطرفية). سنتعلم كل شيء لاحقًا فلا تقلق.

هل تفكر بكتابة شيفرة بسيطة وتنفيذها دون تلك الجاهزة؟ دعنا نفعل ذلك ونكتب شيفرة تطبع رسالة ترحيبية باسمك. نفِّذ الأوامر التالية في سطر الأوامر (بدل `myName` إلى اسمك):

```
$ mkdir -p src/main/kotlin/com/programming/kotlin/chapter01
$ echo "" >> src/main/kotlin/com/programming/kotlin/chapter01/Program.kt
$ cat <<EOF >> src/main/kotlin/com/programming/kotlin/chapter01/Program.kt
package com.programming.kotlin.chapter01
fun main(args: Array<String>) {
     println("Hello myName!")
}
EOF
```

قبـل بنـاء وتنفيـذ الشـيفرة الـتي أنشـأناها للتـو، افتح الملـف build.gradle.kts وعـدل قيمـة mainClassName إلى ما يلي:

```
application {
     // Define the main class for the application.
     mainClassName = "com.programming.kotlin.chapter01.ProgramKt"
}
```

يمكننا الآن بناء وتنفيذ شيفرة التطبيق البسيط عبر الأمرين gradle build وgradle run على التـوالي ويجب أن نحصل على الناتج التالي:

```
Hello myName!
```

ما فعلنا للتو هو ضبط الصنف الذي يحتوي على نقطة دخول البرنامج، وسبب هذا هـو السـماح للبرنـامج للعمـل بشكل مباشرة، ونشرح ذلك قريبًا.

والآن نـرغب في تشـغيل برنامجنا باسـتخدام java -jar [artefact] وقبـل أن نتمكن من فعـل ذلـك، نحتاج إلى تعديل الملف build.gradle ومطابقته. فسنحتاج أولًا إلى إنشاء الملف manifest وتعـيين الصـنف الرئيسي، وذلك لأن JVM سيبحث عن الدالة الرئيسية وسيبدأ في تنفيذها:

```
jar {
     manifest {
             attributes(
                     'Main-Class':
'com.programming.kotlin.chapter01.ProgramKt'
             )
     }
from { configurations.compile.collect { it.isDirectory() ? it :zipTree(it)
} }
}
```

وعلاوة على ذلـــــك، سنضــــمن الاعتماديــــات من أجـــل kotlin-stdlib kotlin-runtime في JAR، لأننـــا إذا لم نفعـــل ذلـــك، فســـنحتاج إلى إضـــافتهم إلى مســـار الصـــنف

(classpath) عند تشغيل التطبيق. والآن، أنت مستعد للبناء وتشغيل الشيفرة البرمجية.

## 6. كوتلن مع Maven

إذا كنت تفضِّل استخدام Maven العجوز، فلا مشكلة في ذلك، فهنالك إضافة لدعم كوتلن أيضًا. إذا لم تكن تملك Maven على جهازك، فانتقل إلى الصفحة maven.apache.org/download.cgi لتنزيله ثم اتبع التعليمات في الصفحة maven.apache.org/install.html لتثبيته على جهازك. سنشرح آلية التنزيل والتثبيت على نسخة أوبنتو، لينكس.

نزل الملف apache-maven-3.6.1-bin.zip (الإصدار الحالي وقت ترجمة هذا الكتاب) ثم فك ضغطه إلى الوجهة `/opt`:

```
unzip apache-maven-3.6.1-bin.zip -d /opt
```

تأكد من متغير البيئة `$JAVA_HOME` ثم أضف مسار maven الذي استخرجناه إليه للتو إلى متغير البيئة `$PATH` بالشكل التالي:

```
$ echo $JAVA_HOME
$ export PATH=/opt/apache-maven-3.6.1/bin:$PATH
```

تأكد بعد ذلك من عمل maven عبر تنفيذ الأمر التالي:

```
$ mvn -v
mvn -v
Apache Maven 3.6.1 (d66c9c0b3152b2e69ee9bac180bb8fcc8e6af555; 2019-04-04T22:00:29+03:00)
Maven home: /opt/apache-maven-3.6.1
Java version: 11.0.2, vendor: Oracle Corporation, runtime: /usr/lib/jvm/java-11-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.15.0-48-generic", arch: "amd64", family: "unix"
```

سنستخدم القوالب المضمَّنة لتوليد مجلد المشروع وتنظيم هيكلية ملفاته مثلما فعلنا مع Gradle. أنشئ

مجلدًا جديدًا باسم HelloWorld (أو أي اسم آخر) ثم انتقل إليه:

```
$ mkdir HelloWorld
$ cd HelloWorld
```

نفِّذ الأمر التالي في الطرفية:

```
$ mvn archetype:generate
```

انتظر تنزيل الملفات الضرورية للمشروع حتى ظهور رسالة ضبط الإعدادات التالية:

```
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 1362:
```

اضغط على زر الإدخال (enter) عدة مرات لاختيار الإعدادات الافتراضية حتى الوصول إلى الخيارين `groupId` و `artifactId` وعندها أدخل القيم التالية لهما:

```
Define value for property 'groupId': com.programming.kotlin
Define value for property 'artifactId': chapter01
```

اضغط على زر الإدخال للخيارات الأخرى لاختيار الإعدادات الافتراضية ليظهر بعدها ملخص حول قيم الخيارات التي أدخلتها. اضغط على زر الإدخال أو `Y` لتأكيد وستنتهي عملية البناء بعدئذ.

```
[INFO] ----------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype:
maven-archetype-quickstart:1.4
[INFO] ----------------------------------------------------------
[INFO] Parameter: groupId, Value: com.programming.kotlin
[INFO] Parameter: artifactId, Value: chapter01
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.programming.kotlin
[INFO] Parameter: packageInPathFormat, Value: com/programming/kotlin
[INFO] Parameter: package, Value: com.programming.kotlin
[INFO] Parameter: groupId, Value: com.programming.kotlin
[INFO] Parameter: artifactId, Value: chapter01
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Project created from Archetype in dir:
/home/userName/HelloWorld/chapter01
[INFO] ----------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time:  35:39 min
[INFO] Finished at: 2019-05-11T02:27:54+03:00
[INFO] ------------------------------------------------------------
```

سيتولد الملف pom.xml والمجلد src اللذين يخصّان Maven في المجلد chapter01 الذي يمثّل المشروع الذي بنيته للتو. لكن قبل إضافة الملف الذي يحتوي على شيفرات برمجية لكوتلن، ستحتاج إلى تفعيل الإضافة. ابدأ بحذف الملف App.java و AppTest.java من src/main/java/com/programming/kotlin و /test/main/java/com/programming/kotlin وأنشئ المجلد src/main/kotlin (يطابق هيكل المجلد الفرعي اسم مجال الاسم):

```
$ cd chapter01
$ mkdir -p src/main/kotlin/com/programming/kotlin/chapter01
$ mkdir -p src/test/kotlin/com/programming/kotlin/chapter01
```

اختر أي محرر وافتح الملف pom.xml، إذ سنعدل عليه وفقًا لما مذكور في الصفحة kotlinlang.org/docs/reference/using-maven.html من توثيق كوتلن الرسمي. تعمل الإضافة kotlin-maven-plugin على تصريف شيفرات كوتلن والإصدار Maven v3 هو المدعوم فقط. شمّر عن ساعديك وابدأ العمل الآن.

عرف إصدار كوتلن الذي تريد استخدامه (ثبّتنا الإصدار 1.3.31 مسبقًا وقد يكون مختلفًا أثناء قراءتك للكتاب، لذا احرص على مطابقته للإصدار المثبت) عبر الخاصية kotlin.version وذلك بنسخ الشيفرة التالية وإضافتها إلى القسم <properties> في الملف pom.xml:

```
<properties>
     <kotlin.version>1.3.31</kotlin.version>
</properties>
```

حدّد المكتبة القياسية المراد استخدامها في تطبيقك وذلك بإضافة الاعتمادية التالية إلى الملف pom.xml

(أضفها داخل <dependencies> دون إعادة نسخ الوسم نفسه مرة أخرى):

```
<dependencies>
      <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>${kotlin.version}</version>
      </dependency>
</dependencies>
```

نأتي الآن إلى ضبط عملية التصريف. نحدّد الملف الجذر (المصدري) للمجلدين اللذين أنشأناهما للتو (إن لم تفعل ذلك، فعد إلى تلك الخطوة ونفذها) من أجل تصريف الشيفرات وذلك بنسخ ما يلي إلى داخل الوسم <build> في الملف pom.xml:

```
<build>
      <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>

<testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirecto
ry>
</build>
```

تحتاج الإضافة أيضًا إلى ضبطها من أجل تصريف شيفرات كوتلن وذلك بإضافة ما يلي إلى داخل <build> أيضًا:

```
<build>
      <plugins>
            <plugin>
                  <groupId>org.jetbrains.kotlin</groupId>
                  <artifactId>kotlin-maven-plugin</artifactId>
                  <version>${kotlin.version}</version>

                  <executions>
                        <execution>
```

```
                    <id>compile</id>
                    <goals> <goal>compile</goal> </goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals> <goal>test-compile</goal> </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

هـذا المشـروع الآن جـاهز لتصـريف وبنـاء شـيفرات كـوتلن فقـط؛ وإن أردت السـماح بخلـط شـيفرات جافـا وتصريفها – إذ ستحتاج في بعض الحالات إلى أن كتابة جزءٍ من الشيفرة البرمجية بلغة جافا - ، فأضف مـا يلي بـدلًا مما سبق:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <goals> <goal>compile</goal> </goals>
                    <configuration>
                        <sourceDirs>

<sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
```

```
<sourceDir>${project.basedir}/src/main/java</sourceDir>
                              </sourceDirs>
                         </configuration>
                    </execution>
                    <execution>
                         <id>test-compile</id>
                         <goals> <goal>test-compile</goal> </goals>
                         <configuration>
                              <sourceDirs>

<sourceDir>${project.basedir}/src/test/kotlin</sourceDir>

<sourceDir>${project.basedir}/src/test/java</sourceDir>
                              </sourceDirs>
                         </configuration>
                    </execution>
               </executions>
          </plugin>
          <plugin>
               <groupId>org.apache.maven.plugins</groupId>
               <artifactId>maven-compiler-plugin</artifactId>
               <version>3.5.1</version>
               <executions>
                    <!-- Replacing default-compile as it is treated
specially by maven -->
                    <execution>
                         <id>default-compile</id>
                         <phase>none</phase>
                    </execution>
                    <!-- Replacing default-testCompile as it is treated
specially by maven -->
                    <execution>
```

```
                    <id>default-testCompile</id>
                    <phase>none</phase>
                </execution>
                <execution>
                    <id>java-compile</id>
                    <phase>compile</phase>
                    <goals> <goal>compile</goal> </goals>
                </execution>
                <execution>
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals> <goal>testCompile</goal> </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

لاحـظ أنَّه إذا إن أردت تسريع عملية البناء والتصريف، يمكنك تفعيل ميزة التصريف التدريجي (incremental compilation) والمدعومة بدءًا من الإصدار 1.1.2 من كوتلن وذلك عبر إضافة الخاصية التالية `kotlin.compiler.incremental` إلى `<properties>`:

```
<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

لقد حان الآن وقت إضافة شيفرة !Hello World، وهذا الخطوة تشبه لخطوة أخرى قمنا بها عندما تحدثنا عن Gradle:

```
$ echo "" >> src/main/kotlin/com/programming/kotlin/chapter01/Program.kt
$ cat <<EOF >> src/main/kotlin/com/programming/kotlin/chapter01/Program.kt
package com.programming.kotlin.chapter01
```

```
fun main(args: Array<String>) {
     println("Hello World!")
}
EOF
```

نستطيع الآن تصريف وبناء ملف JAR للبرنامج عبر تنفيذ الأمرين التاليين:

```
$ mvn package
$ mvn exec:java -
Dexec.mainClass="com.programming.kotlin.chapter01.ProgramKt"
```

يجب أن تنتهي التعليمة الأخيرة بطباعة نص Hello World! على الطرفية بشكل شبيه لما يلي:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< com.programming.kotlin:chapter01
>------------------
[INFO] Building chapter01 1.0-SNAPSHOT
[INFO] --------------------------------
[ jar ]--------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ chapter01 ---
Hello World!
[INFO] ---------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ---------------------------------------------------
[INFO] Total time:  2.113 s
[INFO] Finished at: 2019-05-16T18:35:35+03:00
[INFO] ---------------------------------------------------
```

ويمكنك بالطبع تشغيل البرنامج خارج Maven عن طريق الأمر java، لكننا سنحتاج إلى إضافة مُشغِّل كوتلن الآني إلى مسار الصنف (classpath).

```
$java -cp $KOTLIN_HOME/lib/kotlin-runtime.jar:target/chapter01-1.0-
SNAPSHOT.jar "com.programming.kotlin.chapter01.ProgramKt"
```

إذا أردت تجنب إعداد اعتماديات Classpath عند تشغيل التطبيق، هنالك خيار لتحزيم جميع الاعتماديات

في ملف JAR الناتج وإنتاج ما يسمى بملف jar السمين (fat jar)، ولفعل ذلك ستحتاج إلى إضافة ما يلي (وهي إضافة أخرى) إلى القسم `<plugins>` الذي أضفناه منذ قليل داخل القسم `<build>`:

```
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.6</version>
      <configuration>
            <archive>
                  <manifest>
                        <addClasspath>true</addClasspath>

<mainClass>com.programming.kotlin.chapter01.ProgramKt</mainClass>
                  </manifest>
            </archive>
      </configuration>
</plugin>
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>2.4.3</version>
      <executions>
            <execution>
                  <phase>package</phase>
                  <goals>
                        <goal>shade</goal>
                  </goals>
            </execution>
      </executions>
</plugin>
```

يمكننا الآن تنفيذ أمر تشغيل JAR دون القلق على تعيين مسار الصنف (`classpath`) لأنّنا انجزنا ذلك عن طريق الإضافة:

```
$ java -jar target/chapter01-1.0-SNAPSHOT.jar
```

**ملاحظة**

إن استعملت أحدث إصدار من كوتلن وكان غير الإصدار الذي استعملناه في هذا الكتاب – الذي هو 1.3.31-، فانسخ الشيفرات الموجودة في الصفحة kotlinlang.org/docs/reference/using-maven.html من توثيق كوتلن الرسمي بدلًا من الشيفرات السابقة.

## 7. كوتلن وبيئة التطوير IntelliJ

لا يعد المحرر Vim أو nano خيارًا مناسبًا للجميع لكتابة الشيفرات، إذ يزداد العمل صعوبةً كلما ازداد تعقيد المشروع دون استخدام بيئة تطويرية متكاملة (IDE، اختصار للعبارة Integrated development Environment) التي توفّر ميزات كثيرة تسهّل كتابة الشيفرات والتعليمات البرمجية مثل الإكمال التلقائي، والتحسس الذكي (intelli-sense)، والاختصارات المفيدة، وإعادة تصميم الشيفرة.

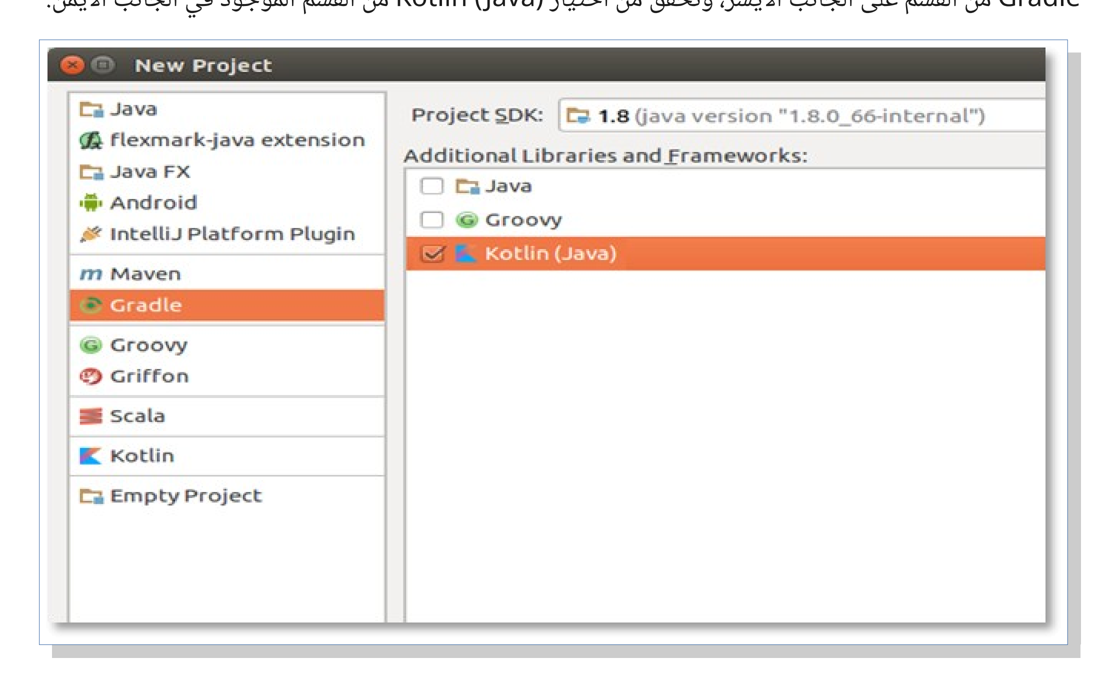
في عالم JVM، كان الخيار الأول للمبرمجين عندما يتعلق الأمر ببيئة تطوير متكاملة (IDE) هو IntelliJ لفترةٍ طويلةٍ، إذ الشركة التي أنشئت هذه البيئة هي الشركة نفسها التي أنشأت كوتلن: JetBrains، وستكون هذه البيئة الخيار الأول بالنسبة لي بالنظر إلى التكامل بين الاثنين لكن، كما سنرى في القسم التالي، فهي ليست الخيار الوحيد.

تأتي بيئة IntelliJ بنسختين: المجانية (Community) والمدفوعة (Ultimate)،وسيكون خيار النسخة المجانيّة كافيًا بالنسبة للشيفرات البرمجية التي سنكتبها خلال رحلتنا مع هذا الكتاب. فإذا لم تكن لديك بالفعل، يمكنك تنزيلها وتثبيها من jetbrains.com/idea/download. إن كنت تستخدم لينكس، توزيعة أوبنتو، يمكنك تثبيتها بسهولة عبر snap بتنفيذ الأمر التالي في الطرفية:

```
sudo snap install intellij-idea-community --classic
```

بدايةً من النسخة 15.0، تأتي بيئة IntelliJ محزمةً مع كوتلن، لكن إذا كنت تمتلك نسخة أقدم من هذه، فيمكنك دعم اللغة عن طريق تثبيت إضافة لها؛ ولفعل ذلك، اذهب إلى Settings ثم Plugins ثم Install IntelliJ plugins واكتب Kotlin في مربع البحث ثم ثبّت الإضافة.

سنستخدم هذه البيئة التطويرية لإنشاء مشروع Gradle مع تفعيل كوتلن. فكما فعلنا في القسم السابق، يجب عليك بمجرد بدء IntelliJ اختيار Create new project وستظهر لك نافذة حوار يمكنك من خلال اختيار Gradle من القسم على الجانب الأيسر، وتحقق من اختيار Kotlin (Java) من القسم الموجود في الجانب الأيمن:



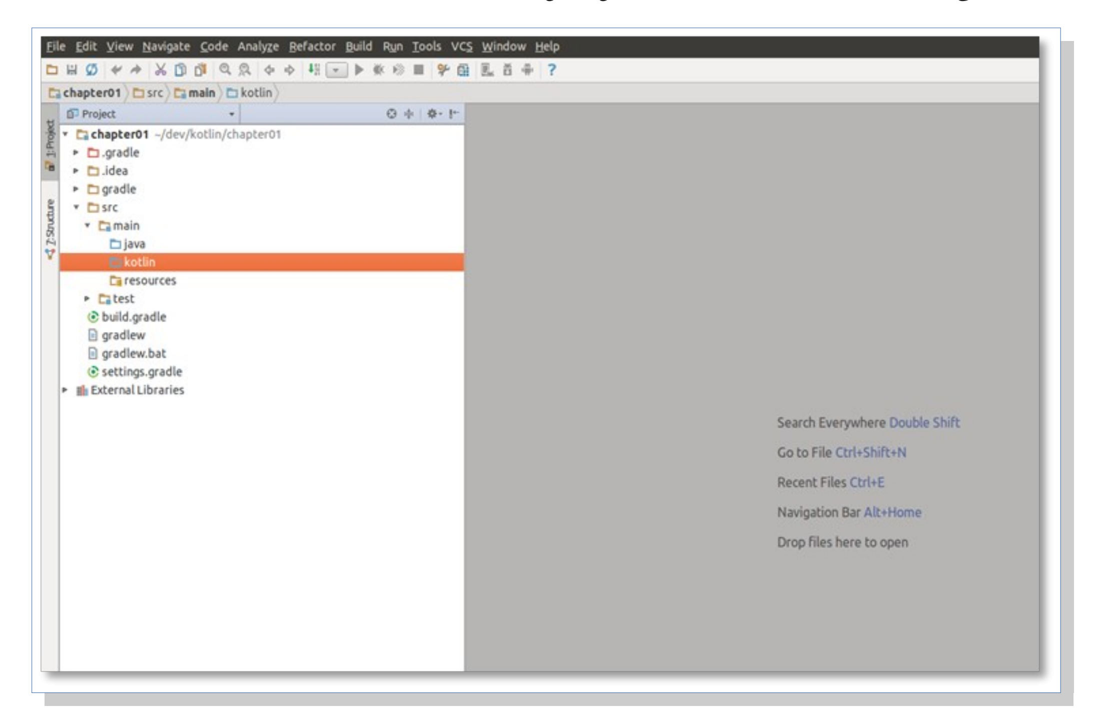

يجب أن تكون قد ضبطت متغير البيئة JAVA_HOME للأداة لتحديد SDK تلقائيًا (انظر إلى الحقل Project SDK في أعلى الصورة التوضيحية). فإذا لم تفعل هذا أو لم تجد شيئًا داخل الحقل Project SDK، فاختر New من القائمة وانتقل إلى حيث توجد جافا JDK الخاص بك وحدّد موقعها. وبمجرد تحديد موقع JDK، فأنت مستعد للانتقال إلى الخطوة التالية عن طريق الضغط على Next الموجود في أسفل يمين النافذة.

تطلب منك النافذة التالية توفير `Group Id` و `Artifact Id`، وهما `com.programming.kotlin` و

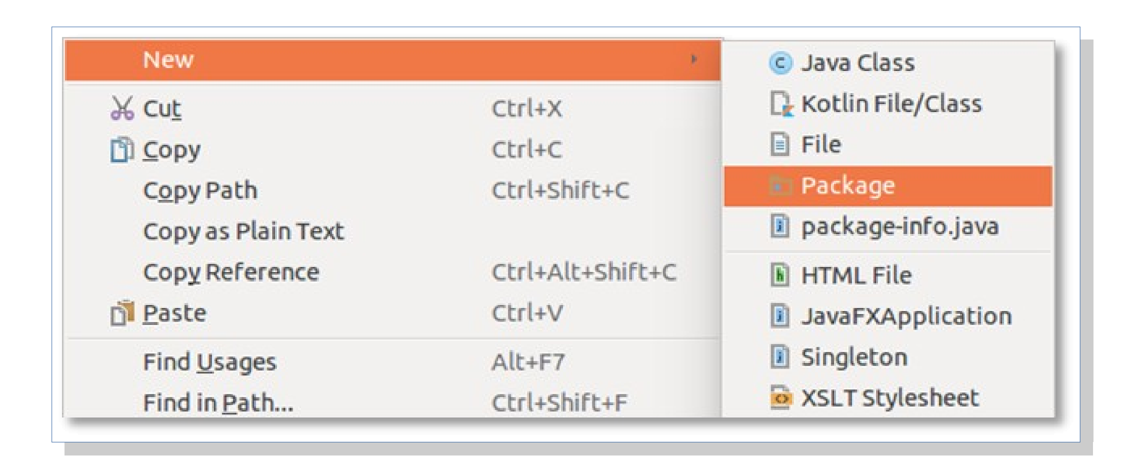
`chapter01` على التوالي. وبمجرد ملئك هذين الحقلين، يمكنك الانتقال للخطوة التالية من العملية وحدّد فيها الحقل `Use auto-import`.

انتقل الآن إلى الخطوة التالية، وهي تحديد مكان تخزين المشروع في جهازك؛ عيِّن مكان المشروع، واضغط على More Settings واكتب chapter01 في الحقل `Module name` واضغط على Finish.

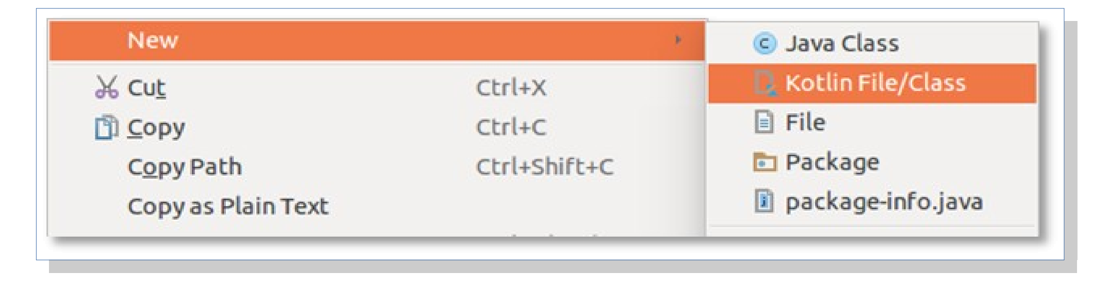
ستنشئ لك IntelliJ مشروعًا جديدًا، ويجب أن تكون النتيجة مشابهة للصورة التالية:



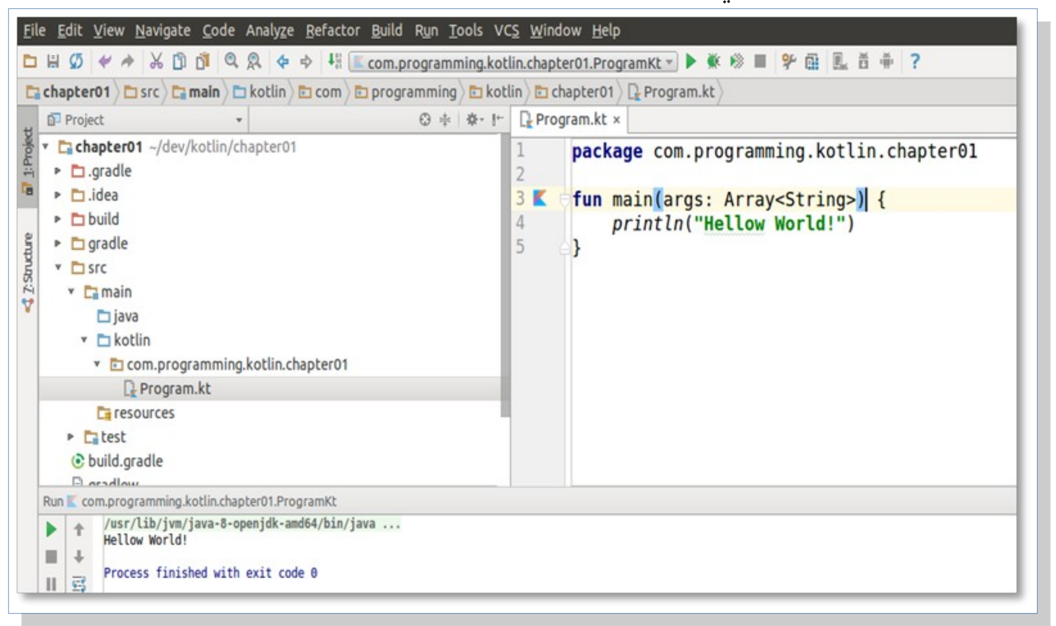

في مجلد kotlin المحدد، انقر بزر الفأرة الأيمن عليه واختر الخيار New ثم Package واكتب `com.programming.kotlin.chapter01`:




سترى في اسفل المجلد kotlin، ظهور مجلد جديد؛ انقر بالزر الأيمن عليه واختر New ثم Kotlin File/Class واكتب Program.kt:

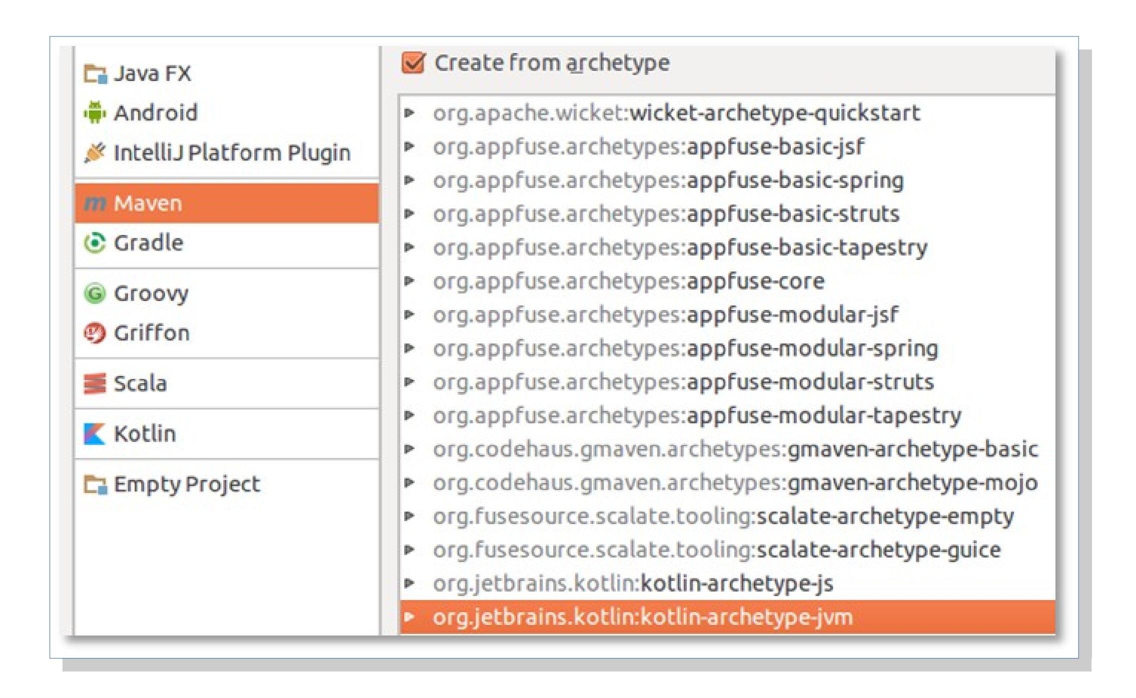


أصبحت الآن مستعدًا لكتابة برنامج Hello World! الخاص بك؛ استخدم نفس الشيفرة البرمجية التي استخدمناها في وقت سابق من هذا الفصل. ستلاحظ أيقونة كوتلن على الجانب الأيسر من المحرر، وإذا نقرت عليه، سيفعّل خيار تشغيل الشيفرة البرمجية. اضغط عليه لتنفيذ الشيفرة (أو اضغط ببساطة في أي مكان من المحرِّر واختر run)، وستُصرَّف الشيفرة آنذاك ثم تُنفَّذ لتظهر أية مخرجات في أسفل نافذة IntelliJ. أي يجب أن ترى العبارة Hello World! مطبوعةً في ذلك القسم:




أحسنت! لقد كتبت أول برنامج كوتلن عبر بيئة تطويرية متكاملة. لابد أنك لاحظت كم كانت عملية إعداد المشروع والشيفرة البرمجية وتشغيلها سهلةً وبسيطةً. يمكنك العمل على Maven بدلًا من Gradle إن أحببت. ما رأيك أن نجرب ذلك؟! هيا بنا.

أنشئ مشروعًا جديدًا من New ثم Project ثم اختر Maven من القسم الأيسر وحدِّد الخيار Create from archetype ثم اختر org.jetbrains.kotlin:kotlin-archetype-jvm من القائمة المعروضة آنذاك:



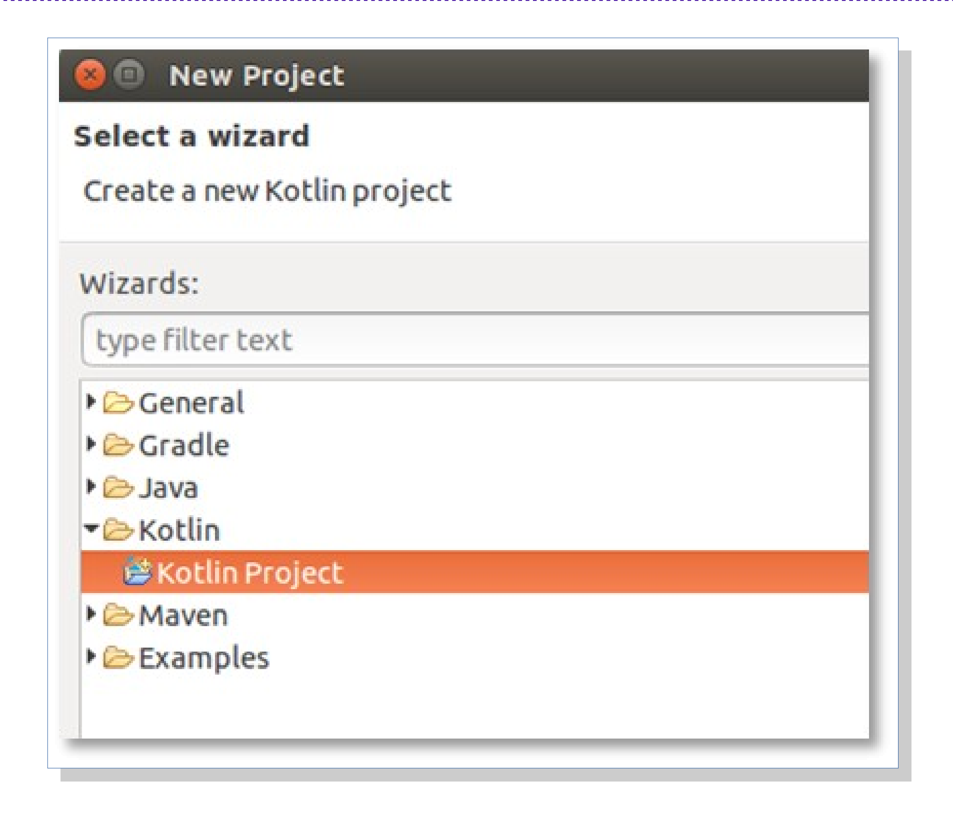

اكمل الخطوات نفسها السابقة وهي كتابة الشيفرة وتنفيذها.

## 8. كوتلن وبيئة التطوير Eclipse

يفضِّل البعض استخدام بيئة التطوير Eclipse المتكاملة بدلًا من البيئة IntelliJ. لا تقلق، فيمكنك كتابة شيفرات كوتلن وتنفيذها دون الحاجة إلى الخروج منه. سأفترض في هذه المرحلة أنك ثبتّ بيئة التطوير هذه (هل تحدّث نفسك بالذهاب إلى الموقع www.eclipse.org/downloads؟! أحسنت الاختيار).

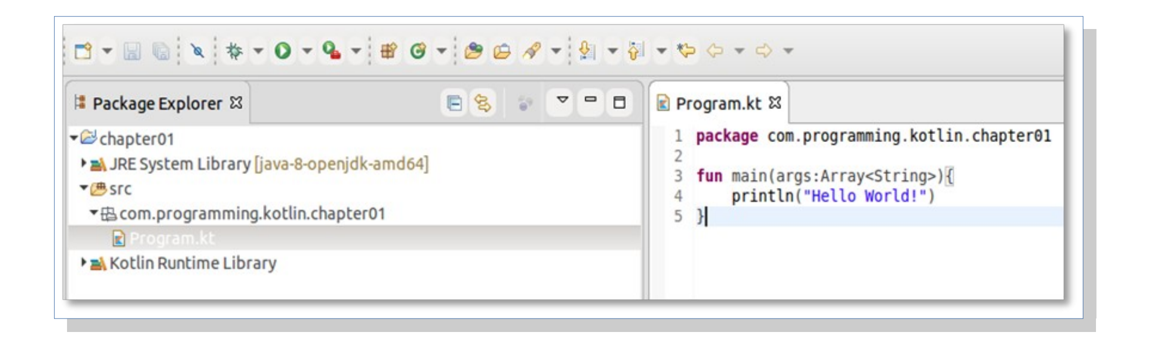
انتقل من القائمة إلى Eclipse Marketplace | Help وابحث عن إضافة كوتلن، وثبتها (أنا استخدم آخر توزيعة: Eclipse Neon).

بمجرد تثبيتك للإضافة وإعادة تشغيل البيئة التطويرية، ستكون مستعدًا لإنشاء أول مشروع كوتلن؛ اختر من القائمة File ثم New ثم Project وسترى نافذة الحوار التالية:




انقر على الزر Next للانتقال إلى الخطوة التالية، وبمجرد اختيارك للشيفرة المصدرية، انقر على الزر Finish؛ هذا ليس مشروع Gradle أو Maven، إذ يمكنك اختيار واحد من هذين، لكنك ستحتاج إلى تعديل الملف build.gradle أو الملف pom.xml يدويًا كما فعلنا في سابقًا مع Gradle و كوتلن مع Maven في هذا الفصل.

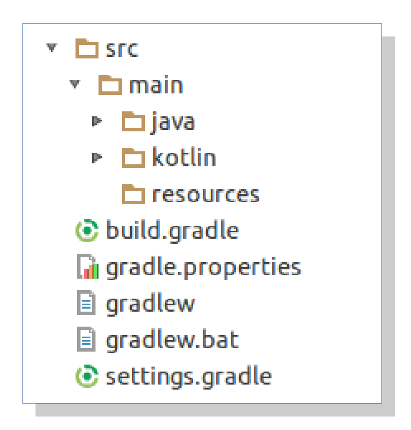
كما في مشروع IntelliJ، انقر على المجلد src واختر New package وسمّ الحزمة الجديدة بالاسم com.programming.kotlin.chapter01؛ اختر بعد ذلك New ثم Other ومن ثم اختر Kotlin ثم Kotlin File من القائمة لإضافة الملف Program.kt؛ وبمجرد إنشاء الملف، اكتب الأسطر البسيطة لطباعة نص على الطرفية:




الآن، أنت مستعد لتشغيل الشيفرة البرمجية؛ اختر من القائمة Run | Run ومن المفترض أن تكون قادرًا على بدء عملية التنفيذ، ويجب أن تظهر لك العبارة !Hello World مطبوعةً في قسم الطرفية في نافذة البيئة التطويرية نفسها.

## 9. الخلط بين كوتلن وجافا في مشروع واحد

من الشائع استخدام لغات برمجة مختلفة في مشروع واحد، لقد عملت على مشاريع حيث تختلط ملفات جافا وScala معًا لتشكيل الشيفرة البرمجية الأساسيّة، فهل نستطيع فعل نفس الشيء مع كوتلن؟ بالطبع. لنعمل على المشروع الذي أنشأناه قبل قليل في القسم 7.1 وهو كوتلن مع Gradle. يجب أن ترى الهيكلية التالية في IntelliJ (القالب القياسي لمشروع جافا/كوتلن):



يمكنك وضع شيفرة برمجية بلغة جافا داخل المجلد java وذلك عن طريق إضافة حزمة جديدة فيه بنفس الاسم الموجود في المجلد kotlin وهو: com.programming.kotlin.chapter01 وإنشاء صنف جافا عبر New ثم Java class باسم CarManufacturer.java واستخدام هذه الشيفرة البرمجية للتمرين:

```
public class CarManufacturer {
   private final String name;
   public CarManufacturer(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
}
```

ماذا لو أردّت إضافة صنف جافا ضمن المجلد الفرعي kotlin؟ لننشئ الصنف Student المشابه للسابق وسنوفِّر اسم الحقل للبساطة:

```
public class Student {
      private final String name;
      public Student(String name) {
            this.name = name;
      }
      public String getName() {
            return name;
      }
}
```

في الدالة main، دعنا ننشئ نسخة من هذا الصنف:

```
fun main(args: Array<String>) {
      println("Hellow World!")
      val student = Student("Alexandra Miller")
      println("Sudent name:${student.name}")
```

```
    val carManufacturer = CarManufacturer("Mercedes")
    println("Car manufacturer:${carManufacturer.name}")
}
```

في حين أنَّ الشيفرة البرمجية تصرَّف بشكل جيِّد، لكن سيُرمَى عند تشغيله "استثناء وقت التشغيل" (runtime exception)، حيث يخبرك أنه لا يستطيع إيجاد الصنف `Student`، ولذلك سنحتاج إلى السماح لمصرّف جافا بالبحث عن الشيفرة البرمجية داخل المجلد src/main/kotlin.

أضف التعليمة التالية في `build.gradle`:

```
sourceSets {
main.java.srcDirs += 'src/main/kotlin'
}
```

يمكنك الآن تصريف البرنامج وتشغيله:

```
$ gradle jar
$ java -jar build/libs/chapter01-1.0-SNAPSHOT.jar
```

عندما تكبر شيفرة كوتلن البرمجية، سيصبح وقت التصريف أبطأ لأنه سيذهب ويعيد تصريف كل ملف، ويمكنك تسريع ذلك عن طريق إعادة تصريف الملفات التي تم تغييرها بين البناءات فقط، وأسهل طريقة لفعل ذلك هي عن طريق إنشاء ملف باسم `gradle.properties` بجانب `build.gradle` وإضافة `kotlin.incremental=true` إليه. وفي حين أن أول عملية بناء لن تصبح تراكمية (أي لن يسري مفعول تفعيل الخاصية `kotlin.incremental` التي فعّلتها)، إلا أنها ستصبح كذلك في عملية البناء الثانية وسترى تحسُّنًا في وقت التصريف.

لا يزال Maven على الأرجح، النظام الأكثر استخدامًا في JVM، لذلك دعنا نرى كيف يمكننا تحقيق هدفنا بخلط شيفرة جافا وكوتلن معًا في Maven.

في IntelliJ، اختر New ثم Project، ثم حدّد Maven كخيار، وابحث عن kotlin-archetype-jvm من قائمة archetypes، ولقد ذكرنا هذا بالفعل في القسم «كوتلن وبيئة التطوير IntelliJ»، لذا يجب أن يكون العملية مألوفةً لديك؛ وبهذا سننشئ مشروعًا جديدًا.

ستلاحظ أنه لم ينشئ مجلد java في شجرة المشروع، لذا أنشئ المجلد src/main/java متبوعًا بمجلدٍ ذي مجال الاسم `com.programming.kotlin` (سيكون هذا مجلدًا فرعيًا للمجلد java).

ستلاحظ أنه لن يتاح لك - عند النقر بزر الفأرة الأيمن - إنشاء حزمة جديدة، فالمشروع غير مهيئ بعد لتضمين شيفرة جافا، لكن هل سألت نفسك قبل كل شيء ما الذي يجعل Maven يتعامل مع شيفرة كوتلن؟ إذا فتحت ملف pom.xml وذهبت إلى قسم الإضافات `<plugins>`، ستلاحظ الإضافة `kotlin`:

```
<plugin>
     <groupId>org.jetbrains.kotlin</groupId>
     <artifactId>kotlin-maven-plugin</artifactId>
     <version>${kotlin.version}</version>
     <executions>
            <execution>
                   <id>compile</id>
                   <phase>compile</phase>
                   <goals>
                          <goal>compile</goal>
                   </goals>
            </execution>
            <execution>
                   <id>test-compile</id>
                   <phase>test-compile</phase>
                   <goals>
                          <goal>test-compile</goal>
                   </goals>
            </execution>
     </executions>
</plugin>
```

ستحتاج لكي تتمكن من خلط شيفرة جافا مع شيفرة كوتلن إلى إعداد إضافة جديدة من أجل تصريف الشيفرات المكتوبة بلغة جافا:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <executions>
        <execution>
            <id>default-compile</id>
            <phase>none</phase>
        </execution>
        <execution>
            <id>default-testCompile</id>
            <phase>none</phase>
        </execution>
        <execution>
            <id>java-compile</id>
            <phase>compile</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
        <execution>
            <id>java-test-compile</id>
            <phase>test-compile</phase>
            <goals>
                <goal>testCompile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

يجب أن يعمل مصرِّف كوتلن قبل مصرِّف جافا حتى يعمل كل شيء جيّدا، لذلك سنحتاج إلى تعديل الإضافة `kotlin-maven-plugin` لفعل ذلك عبر وضع الشيفرة التالية مكانها:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
```

```
        <groupId>org.jetbrains.kotlin</groupId>
        <version>${kotlin.version}</version>
        <executions>
                <execution>
                        <id>compile</id>
                        <goals>
                                <goal>compile</goal>
                        </goals>
                        <configuration>
                                <sourceDirs>

        <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>

        <sourceDir>${project.basedir}/src/main/java</sourceDir>
                                </sourceDirs>
                        </configuration>
                </execution>
                <execution>
                        <id>test-compile</id>
                        <goals>
                                <goal>test-compile</goal>
                        </goals>
                        <configuration>
                                <sourceDirs>

        <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>

        <sourceDir>${project.basedir}/src/main/java</sourceDir>
                                </sourceDirs>
                        </configuration>
                </execution>
        </executions>
</plugin>
            <plugin>
                 <groupId>org.apache.maven.plugins</groupId>
                 <artifactId>maven-compiler-plugin</artifactId>
```

```
                <version>3.5.1</version>
                <executions>
                    <!-- Replacing default-compile as it is treated 
specially by maven -->
                    <execution>
                        <id>default-compile</id>
                        <phase>none</phase>
                    </execution>
                    <!-- Replacing default-testCompile as it is treated 
specially by maven -->
                    <execution>
                        <id>default-testCompile</id>
                        <phase>none</phase>
                    </execution>
                    <execution>
                        <id>java-compile</id>
                        <phase>compile</phase>
                        <goals> <goal>compile</goal> </goals>
                    </execution>
                    <execution>
                        <id>java-test-compile</id>
                        <phase>test-compile</phase>
                        <goals> <goal>testCompile</goal> </goals>
                    </execution>
                </executions>
            </plugin>
```

وستحتاج إلى إضافة أخرى من Maven حتى تتمكن من إنتاج ملف JAR قابل للتنفيذ للشيفرة البرمجية التي نكتبها:

```
<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-jar-plugin</artifactId>
	<version>2.6</version>
	<configuration>
		<archive>
			<manifest>
				<addClasspath>true</addClasspath>

	<mainClass>com.programming.kotlin.HelloKt</mainClass>
			</manifest>
		</archive>
	</configuration>
</plugin>
```

ستولد الشيفرة البرمجية السابقة ملف JAR يحتوي على شيفرتك البرمجية فقط، وستحتاج إذا أردت تشغيله إلى اعتماديات أخرى في مسار الصنف `classpath`:

```
<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-assembly-plugin</artifactId>
	<version>2.6</version>
	<executions>
		<execution>
			<id>make-assembly</id>
			<phase>package</phase>
			<goals> <goal>single</goal> </goals>
			<configuration>
				<archive>
					<manifest>

	<mainClass>com.programming.kotlin.HelloKt</mainClass>
					</manifest>
				</archive>
				<descriptorRefs>
```

```
    <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
            </execution>
        </executions>
</plugin>
```

يمكننا الآن إضافة أصناف من المثال السابق (أصناف `CarManufacturer` و `Student`) وتغيير الصنف الرئيسي إلى التالي:

```
val student = Student("Jenny Wood")
println("Student:${student.name}")
val carManufacturer = CarManufacturer("Honda")
println("Car manufacturer:${carManufacturer.name}")
```

لم ننتهِ بعد، في حين أن التصريف سار على ما يرام، إلا أنه عند تنفيذ ملف JAR سيصدر خطأ أثناء وقت التشغيل بسبب أنه لم يجد صنف `Student`، يجب أن يعرف مصرّف جافا أن شيفرة جافا موجودة داخل مجلد `kotlin`، ولذلك سنحتاج إلى إضافة أخرى وهي:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals><goal>add-source</goal></goals>
            <configuration>
                <sources>

    <source>${project.basedir}/src/main/kotlin</source>
                </sources>
            </configuration>
        </execution>
    </executions>
```

```
</plugin>
```

يجب أن تتمكن الآن من تصريف الشيفرة البرمجية وتشغيلها، إذ ستطبع ثلاثة أسطر كمخرجات عند تشغيل هذه الأوامر في الطرفية:

```
$ mvn package
$ java -jar target/chapter01-maven-mix-1.0-SNAPSHOT-jar-with-
dependencies.jar
```

## 10. خلاصة الفصل

لقد تعلمت في هذا الفصل كيف يمكنك إعداد بيئة التطوير الخاصة بك مع الأدوات المطلوبة لبناء وتشغيل شيفرة كوتلن. الآن أنت قادر على تشغيل وتنفيذ الأمثلة التي ستمر معك في الفصول القادمة من هذا الكتاب بالإضافة إلى تجربة شيفرات كوتلن الخاصة بك.

ستخوض في الفصل القادم في بنيات الأساسيات التي ستستخدمها يوميًا عند البرمجة باستخدام كوتلن.

الفصل الثاني:

# أساسيات كوتلن



حان الآن وقت اكتشاف لبنات كوتلن الأساسيَّة. سيوضح هذا الفصل بعض أوجه التشابه والاختلاف الرئيسيَّة بين كوتلن وجافا، وما هي مميزات لغة كوتلن مقابل تلك الموجودة في جافا وJVM، ويمكنك تجاوز هذه الاختلافات إذا لم تكن لديك أية فكرة عن البرمجة بجافا.

سنغطي في هذا الفصل المواضيع التاليّة:

- المتغيرات والقيم
- تدفق تنفيذ الشيفرة وتعابير التحكم
- استنتاج النوع
- التحويلات الذكية
- الأنواع الأساسيّة و التسلسل الهرمي للأنواع في كوتلن

## 1. القيم والمتغيِّرات

تمتلك كوتلن كلمتين مفتاحيَّتين للإعلان عن المتغيرات وهما val و var، فتعني var (اختصار للكلمة variable) متغير قابل للتغيير، والذي هو متغير يمكن تغيير قيمته عن طريق إعادة تعيينه، ويكافئ استعماله التصريح عن متغير في جافا:

```
var name = "kotlin"
```

صرَّحنا عن متغيِّرٍ اسمه nameوأسندنا قيمة إليه مباشرةً. ويمكنك التصريح عن متغير عبر var ثم تهيئته بإسناد قيمة له فيما بعد:

```
var name: String
name = "kotlin"
```

يمكن تغيير قيمة المتغيّرات التي صُرِّح عنها مع var لأنها قابلة للتغيير (mutable):

```
var name = "kotlin"
```

```
name = "more kotlin"
```

على النقيض، تُستخدَم الكلمة المفتاحية `val` للتصريح عن متغيرات قابلة للقراءة فقط، وهو مشابه للتصريح عن متغير ثابت (عبر الكلمة المفتاحية `final`) في جافا. ويجب تهيئة الثابت الذي يصرح عنه بوساطة `val` بقيمةٍ ما لأنه لا يمكن تغييرها لاحقًا:

```
var name = "kotlin"
```

لا يعني المتغير القابل للقراءة فقط أن النسخة نفسها غير قابل للتغيير أيضًا، فيمكن أن تسمح النسخة للمتغيرات الأعضاء (member variables) أن تتغيّر قيمتها عن طريق الدوال أو الخاصيات، ولكن لا يمكن تغيير قيمة المتغير نفسه أو إعادة تعيينه إلى قيمة أخرى.

## 2. استنتاج النوع

هل لاحظت في القسم السابق أنّنا لم نحدّد نوع المتغيّر عند تهيئته؟ هذا مخالف لجافا التي يجب أن نحدّد فيها بدقة نوع المتغير عند التصريح عنه.

لمّا كانت كوتلن هي لغة صارمة في تحديد الأنواع (أي تدعى strongly typed)، فإننا لا نحتاج دائمًا إلى التصريح عن الأنواع صراحةً، إذ سيحاول المصرِّف معرفة نوع التعبير أو المتغير من المعلومات الموجودة فيه. فاستعمال `val` هو حالة في غاية السهولة بالنسبة للمصرّف لأن النوع سيكون واضحًا من الجانب الأيمن من عملية التصريح، وتسمى هذه الآلية بـ «استنتاج النوع» (type inference)، ويقلل هذا من التكرار مع الحفاظ على نوع آمن وهذا ما نتوقعه من لغة حديثة.

لا تعدُّ القيم والمتغيرات الأماكن الوحيدة التي يمكن استنتاج النوع فيها، فتُطبَّق عملية الاستنتاج هذه لمعرفة نوع المعامل/المعاملات من بصمة الدالة (function signature)، ويمكنك الاستفادة منها أيضًا في الدوال المتكونة من سطر واحد حيث يمكن معرفة نوع القيم المعادة (return) من التعبير الذي تُنفِّذه الدالة كما في المثال التالي:

```
fun plusOne(x: Int) = x + 1
```

من المفيد إضافة تحديد النوع بصريح العبارة في بعض الأحيان ويكون ذلك بالشكل التالي:

```
val explicitType: Number = 12.3
```

## 3. الأنواع الأساسيَّة

واحدة من أكبر التغييرات بين كوتلن وجافا هي أن كوتلن تَعُدُّ أن كل شيء هو كائن؛ فإذا كنت مبرمج جافا، فستعرف أنه يوجد في جافا أنواع أساسيَّة (primitive types) خاصَّة يُتعامل معها تعاملًا مختلفًا عن الكائنات، ولا يمكن استخدامها مثل الأنواع المُعمَّمة (generic types)، ولا تدعم استدعاءات التوابع/الدوال، ولا يمكن إسناد القيمة العدمية `null` إليها؛ يُعدُّ النوع `Boolean` المنطقي خير مثالٍ على تلك الأنواع الأساسية التي نتحدث عنها.

قدَّمت جافا كائنات مُغلِّفة (wrapper objects) يمكنها تغليف الأنواع الأساسية في كائنات للتعامل معها على هذا الأساس. على سبيل المثال، يُغلِّف `java.lang.Boolean` النوع `boolean` من أجل تمييزه بسلاسة أكثر. جاءت كوتلن لتزيل هذه الضرورة بشكل كامل من اللغة من خلال ترقية الأنواع الأساسية إلى كائنات كاملة.

سيُعيِّن مصرِّف كوتلن الأنواع الأساسية (basic types) التي يتعرَّف عليها إلى أنواع JVM الأساسية (أي يعيد تعيينها إلى حيث تنتمي بعد أن كانت كائنات) كلما كان ذلك ممكنًا لأسباب تتعلَّق بالأداء؛ ومع ذلك، يجب في بعض الأحيان أن تكون القيم مُعبَّأة، مثل عندما يكون النوع يقبل قيمة `NULL` أو عند استخدامها في الأنواع المُعمَّمة؛ فالتعبئة (Boxing) تُحوِّل من نوع أساسي إلى نوع مُغلَّف حيث يأخذ مكان النوع عند طلبه من قبل كائن لكنه يُقدِّم نوعًا أساسيًّا.

> **ملاحظة**
> قد لا تستخدم قيمتان مختلفتان معبئتان نفس النسخة، لذا لا يمكن ضمان نتيجة تطبيق المساواة المرجعيَّة على القيم المُعبَّئة (boxed values).

### أ. الأعداد

أنواع الأعداد المدمجة هي:

| النوع | الوصف | الحجم |
|---|---|---|
| Long | عدد كبير | 64 |
| Int | عدد صحيح | 32 |
| Short | عدد صغير | 16 |
| Byte | بايت | 8 |
| Double | عدد عشري مضاعف الدقة | 64 |
| Float | عدد عشري | 32 |

لإنشاء عدد من أحد تلك الأنواع، استخدم إحدى النماذج التالية:

```
val int = 123
val long = 123456L
val double = 12.34
val float = 12.34F
val hexadecimal = 0xAB
val binary = 0b01010101
```

ستلاحظ أن الأعداد الكبيرة (أي ذات النوع long) تتطلّب اللاحقة L والأعداد العشرية (أي ذات النوع float) تتطلّب اللاحقة F، ويُستخدم النوع double (عدد عشري مضاعف) كنوع افتراضي للأعداد العشرية، و int (عدد صحيح) للأعداد الصحيحة وتستخدم الأعداد ستة عشرية (hexadecimal) والثنائيّة (binary) البادئة 0x و 0b على التوالي.

لا تدعم كوتلن الاتساع التلقائي للأعداد، لذلك يجب إجراء تحويل بشكل صريح، فيملك كل نوع عددي دالة تحوّل القيمة إلى نوع آخر. فعلى سبيل المثال، يمكنك تحول عدد صحيح إلى عدد كبير بهذه الطريقة:

```
val int = 123
val long = int.toLong()
```

وبالمثل، يمكنك تحويل عدد عشري (float) إلى عدد عشري مضاعف الدقة (double) عن طريق استخدام دالة toDouble:

```
val float = 12.34F
val double = float.toDouble()
```

المجموعة الكاملة لدوال التحويل بين الأنواع العددية هي: - toByte() - toShort() - toInt() - toLong() - toFloat() - toDouble() - toChar().

تدعم كوتلن العمليّات الثنائية (bitwise) الاعتيادية مثل عمليات الإزاحة نحو اليسار (left shift)، والإزاحة نحو اليمين (right shift)، والإزاحة نحو اليمين للعدد عديم الإشارة (unsigned right shift)، والعمليات المنطقية AND و OR و XOR. وخلافًا لجافا، هذه ليست معاملات مدمجة بل دوال مسماة (named functions) لكن يمكن استدعاؤها مثل العوامل:

```
val leftShift = 1 shl 2
val rightShift = 1 shr 2
val unsignedRightShift = 1 ushr 2

val and = 1 and 0x00001111
val or = 1 or 0x00001111
val xor = 1 xor 0x0000111
val inv = 1.inv()
```

لاحظ أنَّ inv (النفي) ليس معاملًا ثنائيًا، بل هو معامل أحادي، لذلك استدعيناه باستخدام صياغة النقطة في العدد.

## ب. القيم المنطقيَّة

يمثِّل النوع Boolean قيمة منطقيّة قياسية ويدعم عملية النفي (negation)، والعملية AND (تدعى رياضيًا conjunction)، والعملية OR (تدعى disjunction). تُقيّم العمليَّتان AND و OR تقييمًا كسولًا، أي إذا توافق الجانب الأيسر مع البند، فلن يُقيّم الجانب الأيمن وهذا ما يطلق عليه «دارة قصيرة» (short-circuit).

```
val x = 1
val y = 2
val z = 2
val isTrue = x < y && x < z
val alsoTrue = x == y || y == z
```

## ت. المحارف

يمثِّل النوع Char محرفًا[1] واحدًا فقط، وتُستخدَم علامة الاقتباس الفردية للدلالة عليها مثل 'A' أو 'Z'، وتدعم عملية تهريب المحارف التاليّة: \t، و \b، و \n، و \r، و '، و "، و \\، و \$.

يمكن تمثيل جميع محارف اليونيكود (unicode) عن طريق استخدام رقم اليونيكود، فعلى سبيل المثال \u1234.

لاحظ أنَّه لا يُعامل النوع char كرقم، كما في جافا.

## ث. السلاسل النصيَّة

كما في جافا، فالسلاسل النصية (أي النوع String) غير قابلة للتعديل، ويمكن الدلالة عليها عن طريق استخدام علامات الاقتباس المزدوجة أو الثلاثيَّة.

تُنشئ علامة الاقتباس المزدوجة سلسة نصيّة مُهرَّبة؛ في السلسلة النصية المهرَّبة، يجب تهريب المحارف الخاصة مثل السطر الجديد (new line):

```
val string = "string with \n new line"
```

تُنشئ علامات الاقتباس الثلاثيَّة سلسلة نصيَّة خام، ولا حاجة إلى التهريب المحارف الخاصة فيها، أي يمكن تضمين جميع المحارف بارتياح:

1 قلت «محرف» وليس «حرف» عن قصد، لأنَّ الفرق بين الحرف والمحرف هو أنَّ الأول يقتصر على الأحرف الهجائية اللغوية (مثل أ ب ج أو A B C ...إلخ.) بينما يشمل الثاني على الحروف الهجائية والأرقام وعلامات الترقيم وحتى محارف التحكم الغير مطبوعة (مثل محرف السطر الجديد).

```
val rawString = """
raw string is super useful for strings that span many lines """
```

تـوفّر السلاسـل النصيـة دالـة تكـرار يمكن اسـتخدامها في حلقـة for وسـنتحدث عنها في قسـم حلقـة حلقـات التكرار.

## ج. المصفوفات

يمكن إنشاء مصفوفة (array) في كوتلن باستخدام الدالة المكتبية arrayOf():

```
val array = arrayOf(1, 2, 3)
```

ويمكنك أيضًا إنشاء مصفوفة من حجم محدَّد ودالة تُستخدم لتوليد كل عنصر:

```
val perfectSquares = Array(10, { k -> k * k })
```

وبخلاف جافا، لا تعامل كوتلن المصفوفات على أنّها نوع خاص، وهي أحد أصناف التجميعات (collections) النظاميّة. وتوفّر نسـخ المصـفوفات دالـة مكـرّرة (iterator function)، والدالـة size بالإضـافة إلى الدالـة get الدالة set؛ ويمكن الاستعاضة عن الـدالتين get و set بصـياغة القوسـين المعقـوفين (أي [ ]) مثـل اللغـات الـتي تتَّبع نمط لغة سي:

```
val element1 = array[0]
val element2 = array[1]
array[2] = 5
```

لتجنب أنواع التعبئة (boxing type) التي سيتم تمثيلها في نهاية المطاف كأنواع أساسـية (primitives) في JVM، توفر كوتلن أصناف مصفوفات بديلة مُخصَّصـة لكـل نـوع من الأنـواع الأساسـية، وتسـمح هـذه للشـيفرات البرمجية ذات الأداء الحرج باسـتخدام المصـفوفات بكفـاءة كمـا في جافـا، وهـذه الأصـناف هي: ByteArray ، و CharArray ، و ShortArray، و IntArray ، و LongArray ، و BooleanArray ، و FloatArray و DoubleArray.

## 4. التعليقات

لن تكون التعليقات في كوتلن جديدة لمعظم المبرمجين لأنها هي نفسها في جافا، وجافاسكربت، وسي ...إلخ. فتدعم كوتلن التعليقات الكتلية والتعليقات السطرية:

```
// تعليق من سطر واحد
/*
تعليق ممتد
على عدة
أسطر
*/
```

## 5. الحزم

تسمح لنا الحزم (packages) بفصل الشيفرة البرمجية إلى مجالات أسماء، ويمكن أن يبدأ أي ملف بسطر التصريح عن الحزمة التي ينتمي إليها:

```
package com.packt.myproject
class Foo
fun bar(): String = "bar"
```

يُستخدَم اسم الحزمة لإعطائنا اسمًا مؤهلًا بالكامل (fully qualified name [اختصارًا FQN]) لصنف، أو كائن، أو واجهة أو دالة؛ ففي المثال السابق، يمتلك الصنف Foo اسمًا مؤهلًا كاملًا وهو com.packt.myproject.Foo وأمّا دالة المستوى الأعلى bar فتملك اسمًا مؤهلًا كاملًا وهو com.packt.myproject.bar.

## 6. الاستيرادات

إذا أردنا استخدام الأصناف، والكائنات، والواجهات والدوال الواقعة خارج الحزمة التي تنتمي إليها الشيفرة (المُصرَّح عنها في بدايتها)، يجب استيراد الصنف أو الكائن أو الواجهة أو الدالة عبر import بالشكل التالي:

```
import com.packt.myproject.Foo
```

### أ. استيرادات بالجملة

إذا كان لدينا مجموعة من الاستيرادات من الحزمة نفسها، فلا يُعقَل تحديد كل عملية استيراد بشكل فـردي بـل يمكن استيراد كامل الحزمة دفعة واحدة باستخدام المعامل * (ويسمى «محرف بدل» [wildcard] أيضًا):

```
import com.packt.myproject.*
```

ستكون اسـتيرادات محـرف البـدل مفيـدة بشـكل خـاص عنـد وجـود مجموعـة كبـيرة من الـدوال المسـاعدة أو الثوابت المعرّفة في مستوى أعلى ونريد الإشارة إليها دون استخدام اسم الصنف الكامل:

```
package com.packt.myproject.constants
val PI = 3.142
val E = 2.178
```

```
package com.packt.myproject
import com.packt.myproject.constants.*
fun add() = E + PI
```

لاحـظ كيـف أنَّ الدالـة `add()` لا تحتـاج للإشـارة إلى `E` و `PI` باسـتخدام الاسـم المؤهـل الكامـل، لكن يمكن استخدامها كما لو كانت في نطاقها، فالاستيراد عبر حرف البدل يزيل التكرار عند استيراد ثوابت عديدة.

### ب. استيراد مع إعادة التسمية

إذا كان هنالك حزمتان تستخدمان الاسم نفسه، فيمكننا استخدام الكلمة المفتاحية `as` لعمل اسم مستعار لاسم إحـدى الحزمـتين، وهـذا مفيـد عنـد اسـتخدام أسـماء شـائعة في مكتبـات متعـددة مثـل `java.io.Path` و `org.apache.hadoop.fs.Path`:

```
import com.packt.myproject.Foo
import com.packt.otherproject.Foo as Foo2

fun doubleFoo() {
     val foo1 = Foo()
```

```
    val foo2 = Foo2()
}
```

## 7. قوالب السلسلة النصيَّة

لاشك أنَّ مطورو جافا على دراية بكيفية وصل السلاسل النصية مع بعضها بعضًا (string concatenation) لخلط التعابير مع سلاسل نصية صرفة مثل:

```
val name = "Sam"
val concat = "hello " + name
```

قوالب السلسلة النصيّة هي طريقة بسيطة وفعّالة لتضمين القيم أو المتغيرات أو حتى التعابير داخل السلسلة النصية دون الحاجة إلى نمط الاستبدال أو وصل السلاسل النصية كما رأيت آنفًا. تدعم الآن لغات عديدة هذه الميزة، واختار مصممي كوتلن اعتمادها أيضًا (قد يشار إلى هذه التقنية في كوتلن «باستيفاء السلسلة النصية» [string interpolation]).

إن قوالب السلسلة النصيّة في كوتلن تتفوق على جافا عند استخدام عدة متغيرات في قالب سلسلة نصية واحد، وستُبقِي بذلك السلسلة النصية قصيرة وسهلة القراءة.

طريقة استخدامها واضحة للغاية، فيمكن تضمين قيمة أي متغير ببساطة عن طريق وضع البادئة `$` قبله:

```
val name = "Sam"
val str = "hello $name"
```

يمكن تضمين أيّ تعبيرات برمجية عن طريق وضعها ضمن القوسين المعقوصين `${}`:

```
val name = "Sam"
val str = "hello $name. Your name has ${name.length} characters"
```

## 8. المجالات

المجال (range) هو مجموعة من القيم التي لها بداية ونهاية، ويمكن إنشاء مجال من أي نوع قابل للموازنة عن طريق استخدام المعامل `..`:

```
val aToZ = "a".."z"
val oneToNine = 1..9
```

بمجرد إنشاء المجال، يمكن استخدام المعامل in لاختبار ما إذا كانت قيمة معينة واقعة ضمن هذا المجال، ولهذا يجب أن تكون الأنواع قابلة للموازنة. لتكون قيمة مضمنة في المجال، يجب أن تكون أكبر من أو تساوي قيمة البداية وأصغر من أو تساوي قيمة النهاية.

```
val aToZ = "a".."z"
val isTrue = "c" in aToZ
val oneToNine = 1..9
val isFalse = 11 in oneToNine
```

يمكن الاستفادة من المجالات العددية (int، أو long أو char) في الحلقات التكرارية (حلقة for مثلًا ويمكنك الإطلاع على قسم حلقة for للمزيد من التفاصيل).

هنالك دوال مكتبية لإنشاء مجالات لا يمكن شملها باستخدام المعامل .. مثل الدالة downTo() التي تنشئ لك مجالًا عن طريق العد التنازلي والدالة rangeTo() التي تنشئ لك مجالًا يتزايد حتى قيمة محدَّدة، وكلا هاتين الدالتيّن مُلحَق (أي مُعرَّف) بالأنواع العدديّة:

```
val countingDown = 100.downTo(0)
val rangeTo = 10.rangeTo(20)
```

انتبه إلى أنَّ التعديل على أي مجال يولِّد مجالًا جديدًا. يمكنك مثلًا استخدام الدالة step() لتحديد الخطوة في المجال:

```
val oneToFifty = 1..50
val oddNumbers = oneToFifty.step(2)
```

لا يمكنك استخدام قيمة سالبة لإنشاء مجال تنازلي، ويمكن عكس المجال عن طريق استخدام الدالة reversed()، وكما يشير الاسم، فإنه يعيد مجالًا جديدًا مع تبديل قيم البداية والنهاية بالإضافة إلى إلغاء قيمة الخطوة:

```
val countingDownEvenNumbers = (2..100).step(2).reversed()
```

## 9. حلقات التكرار

تدعم كوتلن حلقتي التكرار الشهيرتين، حلقة while وحلقة for، الموجودتين في معظم اللغات، وستكون صياغة حلقة while في كوتلن مألوفة لمعظم المبرمجين، والتي هي تشبه بالضبط أغلب اللغات الشبيهة بلغة سي:

```
while (true) {
     println("This will print out for a long time!")
}
```

تُستخدم حلقة for في كوتلن لتكرار تنفيذ عملية على أي كائن يُعرِّف دالة أو دالة موسّعة مع مُكرِّر (iterator)، وتوفر جميع التجميعات هذه الدالة:

```
val list = listOf(1, 2, 3, 4)
for (k in list) {
     println(k)
}
val set = setOf(1, 2, 3, 4)
for (k in set) {
     println(k)
}
```

لاحظ الصياغة التي تستخدم الكلمة المفتاحية in. يُستخدَم المعامل in دائمًا مع حلقات for؛ فبالإضافة إلى دعم التجميعات (collection)، تُدعَم المجالات العددية إمّا بشكل صريح مباشر أو بشكل ضمني (ضمن متغير يحوي مجال):

```
val oneToTen = 1..10
for (k in oneToTen) {  // تضمين المجال بمتغير
     for (j in 1..5) {  // استعمال المجال مباشرةً دون تغليفه بمتغير
          println(k * j)
     }
}
```

**تنبيه**

يتعامل المصرِّف مع المجالات بشكل خاص، وتُصرَّف إلى حلقات for مفهرسة تُدعَم بشكل مباشر في JVM، وبذلك نتجنب أية انخفاض في الأداء من إنشاء كائنات المكرر.

يمكن استخدام أي كائن داخل حلقة for بشرط أن يحوي على تنفيذ لدالة تسمى iterator (مُكرِّر ) ممَّا يجعل هذه العملية مرنة للغاية. ويجب أن تُرجع هذه الدالة نسخة من كائن يوفر الدالتين التاليتين:

```
fun hasNext(): Boolean
fun next(): T
```

لا يصرُّ المصرّف على أي واجهة (interface) معينة، طالما أنَّ الكائن الذي أُرجِع يحتوي على هاتين الدالتين. على سبيل المثال، في الصنف String القياسي، توفر كوتلن الدالة الملحقة iterator التي تلتزم بتنفيذ المطلوب وبذلك يمكن استخدام السلاسل النصيّة في حلقة for للتكرار تنفيذ عملية ما على كل محرف من محارفها.

```
val string = "print my characters"
for (char in string) {
     println(char)
}
```

تملك المصفوفات دالة ملحقة تسمى indices والتي يمكن استخدامها للتكرار على فهرس عناصر المصفوفة.

```
for (index in array.indices) {
     println("Element $index is ${array[index]}")
}
```

**تنبيه**

يقدم المصرّف دعمًا خاصًا للمصفوفات، وسيصرّف الحلقة المطبَّقة على مصفوفة إلى حلقة for مفهرسة طبيعيّة لتجنب أي بطئ في الأداء مثلما يفعل مع حلقات المجالات تمامًا.

## 10. معالجة الاستثناءات

تكاد تتطابق طريقة معالجة الاستثناءات في كوتلن مع الطريقة التي تعالج فيها في جافا مع اختلاف جوهري

في شيء واحد وهو أنَّ جميع الاستثناءات في كوتلن غير مُتحقَّق منها (unchecked exception)[2].

وللتذكير، إنَّ الاستثناءات المُتحقَّق منها (checked exceptions) هي تلك التي يجب التصريح عنها كجزء من بصمة التابع[3] أو التي يجب معالجتها داخل التابع. ومثال نموذجي لذلك هو الاستثناء `IOException` والذي يمكن رميه بواسطة دوال الصنف `File`، وبالتالي ينتهي الأمر بالتصريح عنه في أماكن عديدة من مكتبات الدخل والخرج IO.

الاستثناءات غير المُحدَّدة (Unchecked exceptions) هي التي لا تحتاج إلى إضافتها إلى بصمات التابع، ومثال شائع على ذلك هو الاستثناء `NullPointerException` الذي يمكن رميه من أي مكان؛ فإذا كان هذا الاستثناء مُحدَّدًا، فستحتاج كل دالة إلى التصريح عن بصريح العبارة. في كوتلن، بما أنَّ جميع الاستثناءات غير مُحدَّدة، فلا تشكل جزءًا من بصمات الدالة.

تشبه عملية معالجة الاستثناءات تلك التي تُجرَى في جافا، وذلك باستخدام كتل `try`، و `catch`، و `finally`؛ فالشيفرة البرمجية التي تريد أن يُتعامَل معها بأمان ستكون في كتلة `try`، ويمكنك إضافة لا شيء أو أي شي إلى الكتلة `catch` للتعامل مع الاستثناءات المختلفة، وستُنفَّذ كتلة `finally` دائمًا بغض النظر عما إذا حدث استثناء أم لا. وكتلة `finally` اختياريّة، لكن على الأقل يجب أن يكون هنالك كتلة `catch` أو `finally`.

في هذا المثال، ترمي الدالة `read()` استثناءً من النوع `IOException`، ولذلك نرغب في التعامل مع هذا الاستثناء المحتمل في الشيفرة البرمجية الخاصة بنا. ففي هذه الحالة، نفترض أنَّه يجب أن يكون مجرى الإدخال (input stream) مغلقًا بغض النظر إذا كانت القراءة ناجحة أم لا، ولذلك غلَّفنا الدالة `close()` في كتلة `finally`:

---

2 الفرق بين الاستثناءات المُتحقَّق منها وتلك الغير مُتحقَّق منها هو أن الأولى يجري التحقُّق منها في وقت تصريف الشيفرة ولن تُصرَّف الشيفرة إن عُثِر على أي استثناء يرميه جزء من الشيفرة، بينما الثانية هي التي تُرمَى في وقت التشغيل أي لا يجري التحقُّق منها أثناء التصريف، لذا لا يطلب المُصرِّف معالجتها أو تحديدها.

3 بصمة التابع (method signature) هي عدد المعاملات المُمرَّرة للتابع ونوع كل معامل، ويمكن أن يملك التابع (أو الدالة) نفسه عدة بصمات.

```
fun readFile(path: Path): Unit {
    val input = Files.newInputStream(path)
    try {
        var byte = input.read()
        while (byte != -1) {
            println(byte)
            byte = input.read()
        }
    } catch (e: IOException) {
        println("Error reading from file. Error was ${e.message}")
    } finally {
        input.close()
    }
}
```

## 11. استنساخ الأصناف

لا شك أنّ عملية إنشاء نسخة من صنف (تدعى هذه العملية instantiating classes) مألوفةٌ للقراء الـذين لديهم خبرة في البرمجة كائنية التوجه، إذ تستخدم هذه الصياغة الكلمة المفتاحية new متبوعةً باسم الصـنف الـذي أنشأناه. وتُبلغُ الكلمة المفتاحية new المُصرِّف أنه يجب اسـتدعاء دالـة بانيـة خاصة (constructor function) لتهيئة النسخة الجديدة.

تزيل كوتلن كل هذا، وتُعامِل استدعاء الدالة البانيـة مثـل معاملتها لأي دالـة عاديّة، باسـتثناء أنَّ الدالـة البانيـة تحمل دومًا اسم الصنف المراد اشتقاق نسخة منه. وسيتيح هذا لكوتلن حـذف الكلمـة المفتاحيـة new بشـكل كامـل، وتُمرّر المعاملات أيضًا بشكل عادي:

```
val file = File("/etc/nginx/nginx.conf")
val date = BigDecimal(100)
```

## 12. المساواة المرجعيَّة والمساواة الهيكليَّة

عند العمل مع لغة تدعم البرمجـة كائنيّـة التوجـه، يوجـد مفهـوميْن للمسـاواة، الأول هـو عنـدما يشـير مـرجعين منفصلين إلى النسخة (instance) نفسها في الـذاكرة، والثـاني هـو عنـدما يكـون الكائـنان نسـختين منفصـلين في

الذاكرة لكنهما يملكان القيمة نفسها؛ فمطور الصنف يحدد ماهية «القيمة نفسها»، فعلى سبيل المثال، قد نطلب من أجل تساوي نسختين من مربع (أي صنف يدعى Square) أن يكون لهما الطول والعرض نفسه بغض النظر عن إحداثيات كلٍّ منها.

لنبدأ بشرح المفهوم الأول للمساواة وهو «المساواة المرجعيّة» (referential equality)[4]؛ فلاختبار ما إذا كان مرجعان يشيران إلى النسخة نفسها، نستخدم المعامل === (إشارة التساوية الثلاثية) للتحقق من التساوي أو المعامل ==! للتحقق من عدم التساوي:

```
val a = File("/mobydick.doc")
val b = File("/mobydick.doc")
val sameRef = a === b
```

قيمة التحقُّق a === b هي خطأ false، فعلى الرغم من أنَّ a و b يشيران إلى الملف نفسه في القرص، إلا أنَّهما نسختان مختلفان للكائن File.

المفهوم الثاني للمساواة يدعى «المساواة الهيكليّة» (structural equality)؛ فلاختبار ما إذا كان لكائنان القيمة نفسها بغض النظر عما إذا كان هذان الكائنات نسختين منفصلين أم يشيران إلى النسخة نفسها، نستخدم المعامل == للتحقُّق من التساوي أو المعامل =! للتحقُّق من عدم التساوي. و تُترجم استدعاءات الدوال هذه إلى استخدام الدالة equals التي يجب تعريفها في جميع الأصناف. ولاحظ أنَّ هذا المعامل يختلف عن المعامل == المستخدم في جافا، إذ يُستخدَم المعامل == في جافا للمساواة المرجعيّة والتي في الغالب ما يتم تجنُّبها.

```
val a = File("/mobydick.doc")
val b = File("/mobydick.doc")
val structural = a == b
```

لاحظ أنّه عند التحقق من المساواة باستعمال المعامل==، أصبح الكائنان متساويين (أي قيمة التحقُّق صحيحة true)، وهذا بسبب أن الكائن File يُعرّف المساواة على أنها قيمة المسار، ويعود الأمر إلى مصمم الصنف لتحديد كيفية تطبيق المساواة الهيكليّة لذلك الصنف.

4 تدعى أحيانًا عملية التحقق من التساوي هذه «التساوي الصارم» (Strict Equality)

**تنبيه**

إنَّ المعامل == آمن للقيم null، أي لا حاجة للقلق إذا كنت تتحقَّق من نسخة null، إذ أنَّ المصرّف سيضيف تحقُّق null لنا.

## 13. الكلمة المفتاحية this

نرغب في أحيانٍ كثيرةٍ الإشارة إلى النسخة الحالية داخل الصنف أو الدالة؛ فعلى سبيل المثال، قد ترغب نسخة باستدعاء تابع مع تمرير نفسها كوسيط، ولفعل ذلك، نستخدم الكلمة المفتاحية this:

```
class Person(name: String) {
    fun printMe() = println(this)
}
```

في مصطلحات كوتلن، يسمى المرجع المشار إليه بوساطة الكلمة المفتاحية this بـ «المستقبل الحالي» (current receiver)، وهذا بسبب أنَّ النسخة هي من استقبلت استدعاء الدالة؛ فعلى سبيل المثال، إذا كان لدينا سلسلة نصيّة وأريد معرفة طولها عبر استدعاء length، فستكون نسخة السلسلة النصية هي «المستقبل» (receiver).

تشير this في أعضاء الصنف (class member) إلى نسخة الصنف (class instance)، وتشير وفي الدوال الملحقة إلى النسخة التي استدعيت تلك الدالة الملحقة معها (أي طُبِّقَت عليها).

### أ. النطاق

في النطاقات المتداخلة (nested scopes)، قد نرغب في الإشارة إلى نسخة خارجية؛ ولفعل هذا، يجب علينا استخدام this بشكل مغاير وذلك عن طريق إضافة تسمية (label)؛ فالتسمية التي نستخدمها هي في العادة هي اسم الصنف الخارجي، لكن هنالك قواعد معقّدة أكثر للدوال والمغلّفات (closures) سنناقشها في الفصل الخامس.

```
class Building(val address: String) {
    inner class Reception(telephone: String) {
        fun printAddress() = println(this@Building.address)
```

```
        }
    }
```

لاحظ أنَّنا احتجنا إلى الدالة `print` لتأهيل الوصول إلى نسخة `Building` الخارجية، وهذا لأنَّ `this` داخل دالة `printAddress()` قد أشارت إلى أقرب صنف حاوي، والذي هو في هذه الحالة `Reception`. ولا تقلق حول الكلمة المفتاحية `inner`، لأنَّنا سنتحدث عنها في الفصل الثالث، البرمجة كائنيَّة التوجه في كوتلن.

## 14. مرئية المتغيرات

لم تصمم جميع الدوال أو الأصناف لتكون جزءًا من الواجهة البرمجية العامة (public API) الخاصة بك، ولذلك، قد ترغب بالإشارة إلى بعض الأجزاء من شيفرتك البرمجيّة على أنَّها شيفرات داخليّة ولا يمكن الوصول إليها من خارج نطاق الصنف أو الحزمة. والكلمات المفتاحية المستخدمة هنا لتحدّد هذا تسمى «مُعدّلات المرئية» (visibility modifiers).

هنالك أربعة أنواع من مُعدّلات المرئية هي: عامة (public)، وداخليَّة (internal)، ومحميَّة (protected)، وخاصَّة (private) . وإذا لم يكن هنالك مُعدّل للمرئية، فسيُستعمَل المُعدّل الافتراضي وهو العام؛ وهذا يعني أنَّ ذلك الجزء مرئيٌّ بالكامل لأي شيفرة برمجية ترغب في استخدامه.

| تنبيه | إن كنت مطور جافا، فستلاحظ اختلاف المُعدّل الافتراضي في كوتلن مع ذلك المستعمل في جافا والذي يحدّد المرئية على مستوى الحزمة (package-level visibility). |
|---|---|

### أ. المرئية الخاصة private

الوصول إلى أي دالة، أو صنف، أو واجهة ذات مستوى أعلى غير متاح من تلك المعرّفة على أنَّها `private` إلا من نفس الملف.

ستكون أي دالة أو خاصية خاصَّة (أي أضيف إليها المُعدّل `private`) داخل الصنف، أو الواجهة، أو الكائن مرئية للأعضاء الآخرين فقط من نفس الصنف، أو الواجهة، أو الكائن.

```
class Person {
```

```
    private fun age(): Int = 21
}
```

هنا، الدالة `age()` غير قابلة للاستدعاء سوى من الدوال الموجودة داخل صنف `Person`.

### ب. المرئية المحمية protected

لا يمكن التصريح عن الدوال، أو الأصناف، أو الواجهات والكائنات ذات المستوى الأعلى على أنّها محميّة عبر `protected`.

أي دالة أو خاصيّة معرّفة على أنّها محمية عبر المُعدّل `protected` داخل صنف أو واجهة ستكون مرئية لأعضاء ذلك الصنف أو تلك الواجهة أو للأصناف الفرعيّة فقط.

### ت. المرئية الخاصة internal

يتعامل المُعدّل `internal` مع مفهوم الوحدة (module)، إذ تُعرَّف الوحدة على أنَّها وحدة Maven أو Gradle أو IntelliJ؛ فأي شيفرة برمجيّة تعيّن على أنَّها `internal` ستكون مرئيّة من الأصناف والدوال الأخرى الموجودة داخل الوحدة نفسها، وستتصرف `internal` على أنها مرئية للوحدة، بدلًا من إتاحة المرئية للجميع:

```
internal class Person {
    fun age(): Int = 21
}
```

## 15. تعابير التحكم بتدفق التنفيذ

التعبير (expression) هو عبارة عن تعليمة (statement) تُقيَّم إلى قيمة، فالتعبير التالي يقيّم إلى القيمة المنطقية `true`:

```
"hello".startsWith("h")
```

ومن ناحيّة أخرى، لا ترجع التعليمة التالية أية قيمة ناتجة، فالسطر التالي هو تعليمة لأنه يُعيِّن قيمة إلى مُتغيِّر لكنَّه لا يقيّم نفسه لأي شيء:

```
val a = 1
```

في جافا، كتـل التحكم في تـدفق تنفيـذ الشـيفرة الشـهيرة مثـل `if...else` و `try..catch` هي تعليمـات، فهي لا تُقيّم إلى قيمة؛ لذلك، من الشائع في جافا، عند استخدام إحدى كتل التحكم هذه، تعيين النتيجـة الناتجـة إلى مُتغيّر هُيِّئ خارج الكتلة:

```
public boolean isZero(int x) {
      boolean isZero;
      if (x == 0)
             isZero = true;
      else
             isZero = false;
      return isZero;
}
```

في كـوتلن، كتـل التحكم في التـدفق مثـل `if...else` و `try..catch` هي تعـابير، وهـذا معنـاه أنّه يمكن إسناد النتيجة مباشرةً إلى قيمة، أو إعادتها من دالة، أو تمريرها كوسيط إلى دالة أخرى.

وتسمح هذه الميزة الصغيرة والقويّة بتقليل من الشيفرات البرمجيّة وجعلها سـهلة القـراءة وتتجنب في الـوقت نفسه استخدام المتغيّرات القابلة للتغيير. فيمكن تجنب الحالة الاعتياديّة للتصريح عن متغير خـارج التعليمـة `if` ثمّ تهيئته داخل أحد أفرع التعليمة بشكل كامل:

```
val date = Date()
val today = if (date.year == 2016) true else false
fun isZero(x: Int): Boolean {
      return if (x == 0) true else false
}
```

يمكنك استخدام تابع مشابه مع كتل `try..catch` بالشكل التالي:

```
val success = try {
      readFile()
      true
```

```
} catch (e: IOException) {
    false
}
```

في المثـال السـابق، سـيحتوي المتغيّـر success على نـاتج الكتلـة try إذا نُفِّذت بنجـاح وإلا سـيحتوي على القيمة التي تعيدها البنية catch والتي هي في هذه الحالة false.

لا يجب أن تكون التعابير في سطر واحد، بل يمكن أن تكون في كتل بالطبع، ويجب، في تلك الحالات، أن يكون السطر الأخير تعبيرًا يُمثِّل القيمة التي ستُقيِّم الكتلة إليها.

**تنبيه**

عند استخدام التعبير if، يجب عليك تضمين جملة else أيضًا، وإلا فلن يعرف المصرّف ما الذي يفعله إذا لم يتحقق شرط if، وفي هذه الحالة سيعرض المصرّف خطأً وقت التصريف (compile time error).

## 16. صياغة العَدم null

طوني هوار، مخترع خوارزميّة الترتيب السريع (QuickSort)، هو الـذي قـدّم مفهـوم «مرجـع العَـدم» (Null reference) سنة 1965 والذي دعاه آنذاك بـ «خطأ المليار دولار». ولسوء الحظ، يجب علينـا التعـايش مـع مراجـع العَدم Null كما هي موجودة في JVM، لكن تقدم لنا كوتلن بعض الدوال لتسهيل تجنُّب بعض الأخطاء الشائعة.

فتجبرك كوتلن على التصريح عن المتغير الذي يمكن إسناده إلى null مع ? بالشكل التالي:

```
var str: String? = null
```

فإذا لم تفعل ذلك، فلن تُصرّف شيفرتك البرمجيّة، فالمثال التالي سينتج خطأً وقت التصريف:

```
var str: String = null
```

تملك كوتلن أكثر من هذا لمكافحة استثناءات مؤشر العَدم، وسنجد المزيد من المعلومات حـول العَـدم و الأمـان من العدَم في الفصل السابع، الأمان من العَدم، والانعكاس والتوصيفات.

التحقق من النوع وتحويل الأنواع: إذا تم التصريح عن مرجع لنسخة على أنها من النوع A بشكل عام لكننا نريد

التأكد ما إذا كان من النوع B بشكل أكثر تحديد، ففي هذه الحالة توفر لنا كوتلن المعامل is، وهو يكافئ المعامل instanceof في جافا:

```
fun isString(any: Any): Boolean {
     return if (any is String) true else false
}
```

إذا كان نوع الهدف المراد التحقُّق منه خطأ (محاولة تحويل سلسلة نصيّة إلى النوع File)، فسيُرمَى خطأ من النوع ClassCastException في وقت التشغيل.

## أ. التحويل الذكي للأنواع

إذا أردنا بعد التأكد من النوع الإشارة إلى المتغير على أنه نسخة من B، فيجب تحويله إليه. في جافا، يجب فعل ذلك بصريح العبارة، مما يولّد تكرارًا في الشيفرة:

```
public void printStringLength(Object obj) {
     if (obj instanceof String) {
            String str = (String) obj
            System.out.print(str.length())
     }
}
```

مُصرّف كوتلن أكثر ذكاءً (دون التقليل من قدر مصرِّف جافا بالطبع :-) )، وسيتذكر إجراء التأكد من النوع لنا، وسيُحوِّل المرجع ضمنيًا إلى نوع أكثر تحديدًا، ويشار إلى هذه العملية «بالتحويل الذكي بين الأنواع» (smart cast):

```
fun printStringLength(any: Any) {
     if (any is String) {
            println(any.length)
     }
}
```

يعرف المصرّف أنه لا يمكن أن نكون داخل كتلة الشيفرة البرمجية إلا لو كان المتغيّر بالفعل نسخة للسلسلة

النصيَّة، وبالتالي سيُنفِّذ عملية تحويل قسرية لنا، وسيسمح لنا بالوصول إلى التوابع المُعرَّفة في النسخة المشتقة من نوع السلسلة النصيّة.

إنَّ المتغيرات التي يمكن استخدامها في عملية التحويل الذكي تقتصر على تلك التي يمكن للمصرّف أن يضمن أنَّها لا تتغيَّر بين وقت اختبار المُتغيِّر والوقت التي تُستخدم فيها آنذاك؛ وهذا معناه أنَّ حقول var والمتغيرات المحلية التي أُغلقَت وتغيَّرت (استُخدمَت في دالة مجهولة أسندت قيمة جديدة إليها) لا يمكن أن تُستخدَم في عملية التحويل الذكي تلك.

يمكن أن يعمل التحويل الذكي على الجانب الأيمن من العمليات المنطقية التي تُقيّم بِكَسَل (lazily evaluated) إذا كان الجانب الأيسر هو اختبار للنوع:

```
fun isEmptyString(any: Any): Boolean {
     return any is String && any.length == 0
}
```

يعرف المصرِّف أنَّه لن يقيِّم الجانب الأيمن في التعبير && إلا إذا كان الجانب الأيسر صحيحًا true، لذلك يجب أن يكون المتغيِّر var من نوع سلسلة نصيّة، ولذلك يجري المصرِّف عملية تحويل ذكيَّة لنا ويسمح لنا بالوصول إلى الخاصيّة length على الجانب الأيمن.

يُطبَّق الأمر نفسه مع التعبير ||، إذ يمكننا اختبار أنَّ المرجع ليس من نوع مُحدَّد على الجانب الأيسر ويجب، إذا لم يكن كذلك، أن يكون من هذا النوع على الجانب الأيمن، لذا يمكن للمصرِّف أن يجري عملية تحويل ذكيَّة على الجانب الأيمن آنذاك:

```
fun isNotStringOrEmpty(any: Any): Boolean {
     return any !is String || any.length == 0
}
```

في هذا المثال، تختبر الدالة إمَّا أنَّنا لا نملك سلسلة نصية، أو، إذا كنا كذلك، فيجب أن تكون فارغة.

### ب. التحويل الصريح للنوع

نستخدم المعامل as لتحويل نوع مرجع إلى نوع محدَّد تحويلًا صريحًا. كما في جافا، سترمي هذه العملية

الاستثناء ClassCastException إن لم يكن بالإمكان إجراء عملية التحويل هذه:

```
fun length(any: Any): Int {
     val string = any as String
     return string.length
}
```

لا يمكن تحويل القيمة العدميَّة Null إلى نوع لم يُعرّف على أنه يقبل القيمة العدميَّة (أي nullable)، لذلك سيرمي المثال السابق استثناءً إذا كانت القيمة Null. ولتحويل قيمة إلى نوع يمكن أن يقبل قيمة عَدميَّة، نصرِّح ببساطة أنَّ النوع المطلوب قابل للعَدم، كما نفعل ذلك مع أي مرجع:

```
val string: String? = any as String
```

تذكر أنَّه إذا فشلت عملية التحويل، فسيُرمَى الاستثناء ClassCastException، وإذا أردنا تجنب هذا الاستثناء، والحصول على القيمة null في حال فشل التحويل، فيمكننا استخدام عامل التحويل الآمن ?as فهذا العامل سيرجع القيمة المُحوَّلة إذا كان النوع المستهدفمتوافقًا وإلا سيرجع القيمة العدمية null. في المثال التالي، ستنجح أول عملية تحويل ولكن ستفشل العملية الثانية وستُسنَد القيمة null إلى المتغيِّر:

```
val any = "/home/users"
val string: String? = any as? String
val file: File? = any as? File
```

## 17. تعبير when

دُعِمَت التعليمة switch التقليدية في العديد من اللغات مثل سي وسي بلس بلس وجافا، لكنها مقيَّدة نوعًا ما. وفي الوقت نفسه، أصبح مفهوم البرمجة الوظيفيَّة (functional programming) لمطابقة الأنماط سائدًا أكثر، ولذلك مزجت كوتلن بين الإثنين، وقدّمت when البديل القوي للتعليمة switch التي لا تدعم مطابقة النمط الكامل.

هنالك شكلين للتعبير when، الأول شبيه بـ switch، وفي يُقبَل وسيط مع مجموعة من الشروط، تُفحَص كل واحدة منها على حدة مع القيمة؛ أما الثاني فهو بدون وسيط، ويُستخدَم كبديل لسلسلة من شروط if...else.

### أ. الشكل الأول: when مع وسيط

أبسط مثال لاستخدام when هو مطابقة مجموعة مختلفة من الثوابت، وفي هذه الحالة سيكون مشابهًا لاستخدام switch الاعتيادي في لغة مثل جافا:

```
fun whatNumber(x: Int) {
     when (x) {
            0 -> println("x is zero”)
            1 -> println("x is 1”)
            else -> println("X is neither 0 or 1")
     }
}
```

لاحظ أنه يجب أن تكون when شاملة، فالمصرّف يفرض عليك أن يكون الفرع الأخير هو else.

| ملاحظة | إذا استطاع المصرِّف أن يستنتج أنَّك استوفيت جميع الشروط الممكنة، فيمكنك في هذه الحالة حذف else، وهذا الأمر شائع مع الأصناف المغلَّقة أو المُعدِّدات (enums)، وستعرف أكثر عنهم في الفصول القادمة. |
|---|---|

يمكن استخدام when مع تعبير على غرار if...else و try..catch، وستكون النتيجة المعادة هي ناتج تقييم الفرع. في المثال التالي، أسنِد التعبير when إلى المتغيِّر valisZero قبل إعادته:

```
fun isMinOrMax(x: Int): Boolean {
     val isZero = when (x) {
            Int.MIN_VALUE -> true
            Int.MAX_VALUE -> true
            else -> false
     }
     return isZero
}
```

وعلاوةً على ذلك، يمكن دمج الثوابت معًا إذا كانت الشيفرة البرمجيّة المراد تنفيذها لها هي نفسها؛ ولفعل ذلك، نستخدم الفاصلة لفصل الثوابت:

```
fun isZeroOrOne(x: Int): Boolean {
      return when (x) {
              0, 1 -> true
              else -> false
      }
}
```

لاحظ أنه في هذا المثال، تم دمج جمل 0 و 1 معًا وأرجعت القيمة المعادة مباشرةً بدلًا من إسنادها لمتغيّر وسيط.

لا نقتصر فقط على مطابقة الثوابت في كل شرط، بل يمكننا استخدام أية دالة تُرجع نوع المطابقة نفسها. تُستدعَى الدالة وإذا كانت النتيجة تطابق القيمة، فسيُقيّم هذا الفرع:

```
fun isAbs(x: Int): Boolean {
      return when (x) {
              Math.abs(x) -> true
              else -> false
      }
}
```

في هذا المثال، استدعينا الدالة `Math.abs`، وإذا كانت النتيجة هي نفسها القيمة المعطاة، فهذلك يدل على أنَّ القيمة المعطاة تلك مطلقة بالفعل وستُعاد القيمة true؛ وخلافًا لذلك، ستكون نتيجة `Math.abs` مختلفة، وهذا يدل على أنَّ القيمة لم تكن مطلقة وستُرجع القيمة `false`.

المجالات مدعومة أيضًا، فيمكننا استخدام المعامل `in` للتحقق ما إذا كانت القيمة ضمن المجال، ويُقيّم الشرط إلى `true` إذا كان الأمر كذلك:

```
fun isSingleDigit(x: Int): Boolean {
      return when (x) {
              in -9..9 -> true
```

```
            else -> false
        }
}
```

لاحظ أنه إذا كانت القيمة موجودة في المجال `[-9, 9]`، فيجب أن تكون رقمًا واحدًا، وستُرجع القيمة `true`، أو ستُرجَع القيمة `false` خلافًا لذلك.

يمكننا بالطريقة نفسها معرفة ما إذا كانت القيمة موجودة في مجموعة مُحدَّدة:

```
fun isDieNumber(x: Int): Boolean {
        return when (x) {
            in listOf(1, 2, 3, 4, 5, 6) -> true
            else -> false
        }
}
```

وفي النهاية، يمكن أن تستخدم `when` التحويل الذكي، إذ تسمح عملية التحويل الذكي، كما ذكرنا سابقًا، للمصرّف أن يتأكد من نوع المتغير وقت التشغيل وعرضه:

```
fun startsWithFoo(any: Any): Boolean {
        return when (any) {
            is String -> any.startsWith("Foo")
            else -> false
        }
}
```

في المثال السابق، صُرِّح عن المعامل مع النوع `Any`، لذلك لا يوجد تقييد حول ما يمكن تمريره كمعامل (مشابه للنوع `object` في جافا)، ونتحقَّق في داخل تعبير `when` ما إذا كان النوع هو سلسلة نصيّة؛ وإذا كان كذلك، فيمكننا الوصول إلى دوال السلسلة النصيّة مثل دالة `startsWith`.

لا توجد قيود على الجمع بين هذه الأنواع المختلفة، فيمكنك مزج التحويل الذكي، `in`، وأي دالة، وأي ثابت التعبير نفسه.

### ب. الشكل الثاني: when بدون وسيط

يُستخدم الشكل الثاني من when بدون وسيط، وهو بديل لجمل if...else، ويجعل هذا الشيفرة البرمجيّة أوضح خاصةً إذا كانت الشروط هي موازنات بسيطة. يوضح المثال التالي طريقتين لكتابة الشيفرة البرمجية نفسها: الأولى مع كتل if...else التقليديّة والثانية باستخدام when:

```
fun whenWithoutArgs(x: Int, y: Int) {
     when {
            x < y -> println("x is less than y")
            x > y -> println("X is greater than y")
            else -> println("X must equal y")
     }
}
```

## 18. الدالة التي تعيد شيئًا

إن أدرت إرجاع قيمة من دالة، فاستخدم الكلمة المفتاحية return مع القيمة أو التعبير الذي تريد إرجاعه:

```
fun addTwoNumbers(a: Int, b: Int): Int {
     return a + b
}
```

لاحظ أنّنا حددنا القيمة التي ستعيدها الدالة؛ فبشكل افتراضي، تخرج return من أقرب دالة تحيط بها أو من دالة مجهولة؛ لذلك، في الدالة المتداخلة، سيُخرَج من الدالة الداخليّة تلك فقط:

```
fun largestNumber(a: Int, b: Int, c: Int): Int {
     fun largest(a: Int, b: Int): Int {
            if (a > b) return a
                   else return b
     }
     return largest(largest(a, b), largest(b, c))
}
```

في هذا المثال، ترجع الدالة المتداخلة largest نفسها، وإذا كانت الدالة الداخليّة هي دالة مجهولة، فستبقى موجودة:

```
fun printLessThanTwo() {
     val list = listOf(1, 2, 3, 4)
     list.forEach(fun(x) {
            if (x < 2) println(x)
            else return
     })
     println("This line will still execute")
}
```

إذا أردنـــا إرجـــاع قيمـــة من مُغلّف (closure)، فســـنحتاج إلى تهيئـــة `return` باســـتخدام تســمية (label) وإلا ستُعاد القيمة للدالة الخارجيّة. والتسمية هي سلسلة نصية تنتهي بـ @:

```
fun printUntilStop() {
     val list = listOf("a", "b", "stop", "c")
     list.forEach stop@ {
            if (it == "stop") return@stop
            else println(it)
     }
}
```

لا نحتاج إلى تحديد التسمية، فيمكنك استخدام تسمية ضمنيّة التي هي اسم دالة تقبل مُغلّفًا (closure). إذا أعلن عن تسمية، فلن تُولّد تسمية ضمنيّة:

```
fun printUntilStop() {
     val list = listOf("a", "b", "stop", "c")
     list.forEach {
            if (it == "stop") return@forEach
            else println(it)
     }
}
```

## 19. التسلسل الهرمي للنوع

يسمى النوع الأعلى في كوتلن بـ Any، وهذا مماثل للنوع object في جافا. ويُعرّف النوع Any التوابع

الشهيرة toString، و hashCode وequals. ويُعرّف أيضًا التوابع الملحقة apply، و let، و to، وسنشرح هـذه التوابع بالتفصيل في الفصل الخامس، الدوال الأعلى مرتبةً والبرمجة الوظيفيّة.

يشبه النـوع Unit النـوع void في جافـا، فوجـود النـوع Unit هـو أمـر شـائع في لغـات البرمجـة الوظيفيـة، والفرق بينه وبين void صغير جدًا؛ فالنوع Void ليس نوعًا، بـل حالـة حديّة خاصـة (special edge) تُسـتخدم للإشارة إلى المُصرّف على أنَّ الدالة لا تُرجع قيمة، أمَّا Unit فهو نوع صحيح، مع نسخة مفردة واحدة، ويشار إليه بـ Unit أو ( ) . وعند تعريف دالة على أنها ترجع Unit، فسترجع نسخة مفردة من unit.

ويؤدي هذا إلى صلابة في النوع system، إذ أنَّ جميـع الـدوال يمكن تعريفهـا على أنهـا تُرجـع قيمـة، حـتى لـو كانت من النوع Unit، والدوال التي لا تقبل وسائط يمكن أن تُعرّف على أنها تقبل وسائط من النوع Unit.

تختلف كـوتلن عن جافـا بشـكل خـاص عن طريـق إضـافة نـوع سـفلي (bottom type)، يـدعى Nothing، والذي هو نوع لا يملك أية نسخ مُشتقَّة منه. وبشكل مشابه لكون الصنف Any أعلى صـنف (superclass) لجميـع الأنواع، فإنَّ الصنف Nothing هو صنف فرعي (subclass) لجميع الأنـواع. وإن كنت جديـدًا على مفهـوم النـوع السفلي، قد يكون من الغريب أن يكون هنالك نوع مثل هذا النوع، لكن هنالك عدة حالات عملية يفيد فيهـا اسـتخدام مثل هذا النوع.

فأولًا، يمكن استخدام Nothing لإعلام المصـرّف أنَّ الدالـة لا تكتمـل بشـكل طـبيعي؛ فعلى سـبيل المثـال، قـد تدور في حلقة التكرار للأبد أو ترمي دائمًا استثناء. إليك مثال آخر عن تجميعـات فارغـة غـير قابلـة للتعـديل، فيمكن تعيين قائمة فارغـة من Nothing إلى مرجـع يقبـل قائمـة من سلاسـل نصـية، ولأن القائمـة غـير قابـل للتعـديل، فلا يوجد خطر من إضافة سلسلة نصية إلى قائمـة كهـذه؛ وبالتـالي، يمكن إخفـاء وإعـادة اسـتخدام هـذه القيم الفارغـة، وهذه هي أساسيات عمل دالتي المكتبة القياسيّة emptyList() و emptySet() وغيرها من الدوال.

## 20. خلاصة الفصل

قدمت كوتلن العديد من التحسينات على جافا وأبقت العديد من المميزات التي جعلت من جافا واحدة من أكثر اللغات شعبيَّة على مدى العقدين الماضيين. وبعد قراءة هذا الفصل، يجب أن تشعر بالراحة والاطمئنـان بتعلم كـوتلن والغوص في التعرُّف على بعض التحسينات الإنتاجيّة التي تقدمها.

الفصل الثالث:

# البرمجة كائنيَّة التوجه في كوتلن



كوتلن هي لغة برمجة كائنيَّة التوجه (object-oriented programming وتختصر إلى OOP) وتدعم الدوال ذات الترتيب الأعلى (higher-order functions) وتعابير لامدا (lambdas). إذا كنت لا تعرف ما هي لامدا، فلا تقلق، هنالك فصل كامل مخصص لها، وإذا كنت تستخدم لغة وظيفيَّة بالفعل، فستجد أنَّ كوتلن تدعم بنيات مشابهة للغات الوظيفية.

بمرور الزمن، ازداد تعقيد البرامج، وسمحت لنا البرمجة كائنية التوجه بعمل نموذج للمشكلة التي نريد حلها بوصفها كائنات (objects)، إذ يمكنك رؤية كل كائن على أنه حاسوب صغير: يملك حالة (state) ويمكنه تنفيذ شيء ما أو عدة أشياء (actions)؛ فيمكن للكائن من خلال تنفيذ الإجراءات المتاحة إظهار نوع من السلوك، ولذلك هنالك تشابه واضح بين الكائنات/الكيانات والحياة الواقعية.

وُضعت أول صفة من التجريد الكائني التوجه من قبل آلان كي (Alan Key)، وهو من مخترعي أول لغة برمجة كائنية التوجه: Smalltalk، وفي كتابه The Early History Of Smalltalk، وضع النقاط التالية:

- كل شيء هو كائن: الكائن ليس سوى كتلة من الذاكرة حُجزَت وهُيِّئت وفقًا لتصميم/تعريف مُحدَّد. انطلاقًا من مساحة المشكلة التي يجب عليك حلّها، تأخذ كل الكيانات المنطقيَّة وتُترجمها إلى كائنات في برنامجك.
- تتواصل الكائنات عن طريق إرسال وتلقي الرسائل (بمصطلحات الكائنات): سيكون برنامجك عبارة عن مجموعة من الكائنات التي تُنفِّذ إجراءات مختلفة نتيجةً لاستدعاء توابع يعرِّفها كل واحد منها.
- تملك الكائنات ذاكرة خاصة بها (بمصطلحات الكائنات): أي أنه يمكنك إنشاء كائن من خلال تجميع كائنات أخرى.
- كل كائن هو نسخة لصنف (والذي يجب أن يكون كائنًا): فكر في الصنف على أنَّه مخطط تفصيلي يُحدِّد ما يمكن أن يفعله هذا النوع.
- يحتفظ الصنف بالسلوك المشترك لنُسخِه (في شكل كائنات في قائمة البرنامج): هذا يعني أنَّ جميع الكائنات من نوع معين يمكن أن تتلقى الرسائل نفسها؛ بكلمات أخرى، تعرض التوابع نفسها.

توفِّر كوتلن الدعم الكامل للنقاط أعلاه وتدعم أيضًا دعمًا كاملًا الركائز الثلاثة لأي لغة حديثة كائنية التوجه وهي: التغليف (encapsulation)، والوراثة (inheritance)، والتعددية الشكلية (Inheritance). أمَّا

التغليف، فهو معاملة مجموعة من الحقول والتوابع ذات الصلة على أنّها كائن. وأمّا الوراثة، فتصف القدرة على إنشاء صنف جديد من آخر موجود. وأمّا التعددية الشكلية، فيعني أنه يمكنك استخدام أصناف مختلفة بالتبادل رغم أنّ كل واحد يُنفّذ توابعه تنفيذًا مختلفًا. في هذا الفصل، سنتحدث قليلًا حول كيفية دعم لغة كوتلن كل ما سبق.

إن الهدف من البرمجة كائنية التوجه هو مساعدتنا على التخفيف من المشكلات التي تواجهها مع الشيفرات البرمجية الكبيرة، إذ يمكنها أن تسهل علينا فهم الشيفرات البرمجية وصيانتها وتطويرها والحفاظ عليها خالية من الأخطاء من خلال تزويدنا بما يلي:

- البساطة: تحاكي كائنات البرنامج العالم الواقعي مما يقلل من التعقيد ويبسط هيكل البرنامج.
- الجزئية: تأتي الأعمال الداخليّة في كل كائن من أجزاء أخرى من النظام.
- قابلية التعديل: لا تؤثر التغييرات داخل كائنٍ على أي جزء آخر من البرنامج إذا كنت قد صممّت النظام بشكل صحيح.
- قابلية التوسعة: تتغير متطلبات الكائنات كثيرًا، ويمكنك الاستجابة لهم بسرعة عن طريق إضافة كائنات جديدة أو تعديل أخرى موجودة.
- قابلية إعادة الاستخدام: يمكنك استخدام الكائنات في برامج أخرى.

ستتعلّم في هذا الفصل:

- كيفيّة تعريف واستخدام الأصناف والواجهات.
- متى تختار الواجهات بدلًا من الأصناف المجرّدة.
- متى تختار الوراثة بدلًا من التركيب.

## 1. الأصناف

الأصناف (classes) هي اللبنات الأساسيّة لأي لغة برمجة كائنية التوجه. أول من درس مفهوم الصنف هو أرسطو، حيث أنه أول من جاء بفكرة تصنيف الأسماك والطيور، فجميع الكائنات، على الرغم من كونها فريدة من نوعها، هي جزء من صنف وتشترك في سلوك مشترك.

يُمكنك عن طريق الصنف إنشاء نوع خاص بك عن طريق تجميع التوابع والمتغيرات من الأنواع الأخرى. فكر

في الصنف على أنَّه مخطط، فهو يصف البيانات وسلوك النوع.

يُعلن عن الأصناف باستخدام الكلمة المفتاحيَّة class، كما هو موضَّح في المثال التالي:

```
class Deposit {
}
```

موازنةً بجافا، يمكنك تعريف عدة أصناف داخل الملف المصدري نفسه، فيسبق مستوى الوصول الكلمة المفتاحيّة class، وإذا لم يُحدد مستوى الوصول، فالحالة الافتراضيّة هي المرئيَّة العامة (public)، وهذا يعني أن أي شخص يمكنه إنشاء كائنات من هذا الصنف. يأتي بعد الكلمة المفتاحية class اسم الصنف وبعده القوسين المعقوصين {} اللذين يحتويان على جسم الصنف الذي يُعرِّف فيه السلوك والبيانات: الحقول والخاصيات والتوابع.

يدعم هيكل الصنف الخاصيّة الأولى للغة كائنية التوجه وهي: التغليف. الفكرة وراء ذلك هو أنك تريد إبقاء كل صنف محصّن ومكتفي ذاتيًا، ويسمح لك هذا بتغيير طريقة التنفيذ دون التأثير على أي جزء من الشيفرات البرمجيّة التي تستخدمها، طالما أنَّها تواصل وفاءها لشروط العقد.

حتى الآن، استخدمت مصطلحي الصنف و الكائن بشكل متبادل، وسوف نفهم مع الوقت كيف نميّز بينهما؛ فالكائن هو نسخة من هيكل الصنف وتعريفه وقت التشغيل؛ ولإنشاء نسخة من صنف، تحتاج إلى استدعاء الباني (constructor). ففي المثال السابق، يحصل الصنف Deposit على بانٍ فارغ يولده المُصرِّف تلقائيًا، ولكن إذا أردت تعريف الباني وتوفيره يدويًا، فستحتاج إلى كتابته بالشكل التالي:

```
class Person constructor(val firstName: String, val lastName: String, val
age: Int?) {}

fun main(args: Array<String>) {
   val person1 = Person("Alex", "Smith", 29)
   val person2 = Person("Jane", "Smith", null)
   println("${person1.firstName},${person1.lastName} is ${person1.age}
years old")
   println("${person2.firstName},${person2.lastName} is $
{person2.age?.toString() ?: "?"} years old")
}
```

إذا كنت مطور جافا خبير، فمن المرجّح أنك لاحظت عدم وجود الكلمة المفتاحية new، فلإنشاء نسخة جديدة لصنف معيّن في جافا، تستخدم دائمًا new MyClass، أمّا في كوتلن، فلا تحتاج إلى استخدامها، وإذا فعلت ذلك، فسيخبرك المصرّف بوجود خطأ ناتج عن عدم تعرُّفه على هذه الكلمة المفتاحيَّة الغريبة!

بالنسبة لمطوّر سكالا، ستبدو الشيفرة البرمجيّة السابقة مألوفةً للغاية. رغم أنَّك ستسأل عن سبب استخدام الكلمة المفتاحيَّة constructor، ألا يعلم المصرّف أنه في سياق الباني؟ الجواب هو أنك لست بحاجة لاستعمالها ما لم تحدد وصول المغيِّر أو توصيفات. يسمى الباني السابق بالباني الأولي، وأتوقع أن سؤالك التالي سيكون: كيف يمكن للباني الأولي أن يحتوي على شيفرة مصدريّة؟ على أي حال، هل تريد التحقق من صحة المعاملات الواردة؟ الجواب داخل كتلة init. لجعل أي شيفرة برمجية تعمل كجزء من بانيك الرئيسي، يجب عليك القيام بما يلي:

```
class Person (val firstName: String, val lastName: String, val age: Int?){
   init{
      require(firstName.trim().length > 0) { "Invalid firstName
argument." }
      require(lastName.trim().length > 0) { "Invalid lastName argument." }
      if (age != null) {
         require(age >= 0 && age < 150) { "Invalid age argument." }
      }
   }
}
```

ستعمل الشيفرة التحقق بوصفها جزءًا من الباني الرئيسي، وسيرمي التابع المطلوب الاستثناء IllegalArgumentException مع الرسالة التي قدّمتها إذا كان تعبير التحقق يساوي False.

أنا متأكد من أن أحدًا من القرّاء يتساءل عن كيفية عمل هذا مع تلك الوسائط الثلاثة، فهل تم إنشائها بعدّها حقولًا عامة للصنف؟ الجواب هو لا، فهي خاصّيّات (properties)، وإذا كنت آتيًا من عالم NET.، فستعرف مباشرةً كل ما يعينه هذا. وسنناقش في فصل لاحق كيف تعمل الخاصيات.

كيف يمكن للمرء إنشاء نسخة جديدة من الصنف Person وجلب القيم من الحقول الثلاثة عند استخدام

الصنف من الشيفرة البرمجية المكتوبة بلغة جافا؟ يجري ذلك من خلال دوال الجلب التي اعتاد عليها أي مطور جافا:

```
Person p = new Person("Jack", "Miller", 21);
System.out.println(String.format("%s, %s is %d age old", p.getFirstName(),
p.getLastName(), p.getAge()));
```

المعامل الثالث من الباني هو عدد صحيح قابل للعَدم (nullable)، وسيكون من جيد أن نحصل على خيار `null` عند اشتقاق نسخة من الصنف دون تمرير قيمة للعمر.

كوتلن لغة برمجة حديثة وتدعم القيم الافتراضية لمعامل التابع، لكن لنفترض في هذه الحالة أنها لا تفعل ذلك، لذلك نريد الحصول على بانٍ ثانٍ نمرر له الاسم الأول `firstName` واسم العائلة `lastName`:

```
constructor(firstName: String, lastName: String) : this(firstName,
lastName, null)
```

ستحتاج إلى استدعاء الباني الرئيسي عن طريق `this` ضمن أي بانٍ ثانوي، ثم مرِّر جميع المعاملات المطلوبة إليه. يمكنك الآن إنشاء كائن `Person` جديد بالشكل التالي:

```
val person2 = Person("Jane", "Smith")
```

إذا كنت لا ترغب في الوصول إلى الباني مباشرةً، فالجأ إلى المرئية المحمية (protected) أو الخاصة (private) أو الداخلية (internal).

يتألف التصميم المنفرد النموذجي من بانٍ خاص مع التابع `getInstance()` الذي ينشئ لك نسخة واحدة من ذلك الصنف وقت التشغيل. يجب عليك عند تعريف أصناف مجرّدة تحديد مستوى المرئية إلى المرئية المحمية، ويمكن بهذه الطريقة استدعاؤها من قبل الأصناف المشتقّة فقط، وسنرى هذا قريبًا بعد أن نتحدث عن الوراثة.

حسب منطق الوحدة الخاصة بك، يمكنك كشف الأصناف التي يمكن ويجب إنشاء نسخ منها داخل وحدتك فقط:

```
class Database internal constructor(connection:Connection) {
}
```

لا يعدُّ وضع بادئة لوسائط الباني مع val أو var أمرًا ضروريًا؛ فإذا لم ترغب بتوليد جالب (أو ضابط إذا استخدمت var)، يمكنك دائمًا فعل ما يلي:

```
class Person2(firstName: String, lastName: String, howOld: Int?) {
   private val name: String
   private val age: Int?

   init {
      this.name = "$firstName,$lastName"
      this.age = howOld
   }

   fun getName(): String = this.name

   fun getAge(): Int? = this.age
}
```

جرّب إنشاء نسخة جديدة للصنف ومن ثمَّ استخدام معامل النقطة ليعرض عليك المُتحسِّس الذكي (intelli-sense) التوابع المتاحة في كائنك[5].

على عكس المثال الأول، لن تُترجم المعاملات الثلاثة إلى حقول: ستعرض النافذة المنبثقة تابعين باسم getName و getAge.

## أ. مستويات الوصول

تملك جميع الأنواع وأعضاء الأنواع (type members) مستويات لإمكانية الوصول (accessibility levels)، الأمر الذي يقيد المكان الذي يمكن أن يُستخدمَوا فيه. كما ذكرنا مسبقًا، إذا لم تحدِّد مستوى وصول، فيُحدَّد مستوى الوصول الافتراضي إلى «العام» (public)، وتأتي كوتلن مع ثلاثة مستويات وصول، وهي:

- داخلي internal: يعني هذا أنه يمكنك إنشاء نسخة جديدة لصنفك من أي مكان من داخل وحدتك.

5 هذا متاح فقط في بيئات التطوير المتكاملة (IDEs) التي تدعم التحسس الذكي مثل بيئة التطوير IntelliJ IDEA على سبيل المثال لا الحصر.

- خاص `private`: هذا أكثر تقيدًا من سابقه لأن الصنف مرئي فقط في نطاق الملف الذي يُعرِّفه.
- محمي `protected`: يمكنك استخدام مستوى الوصول هذا فقط في الأصناف الفرعيّة، وهو غير متاح للأنواع المُصرَّح عنها على مستوى الملف (file-level type of declaration).

يعادل مستوى الوصول الداخلي `internal` مستوى الوصول الخاص `private` للأصناف عندما يتعلق الأمر بالتغليف (encapsulation) في هذه المرة فقط على مستوى الوحدة؛ ويمكنك جعله مرئيًّا على مستوى الوحدة فقط إذا لم يتم الوصول إلى الشيفرة المصدرية من خارج نطاق الوحدة. ويقلل هذا من الواجهة البرمجية (API) التي تنشرها ويجعلها أسهل للفهم؛ وعلاوةً على ذلك، إذا كان هنالك تغيير مطلوب لوحدتك، فيمكنك أن تفترض أنَّ تعديل العقد سيؤدي إلى كسر الواجهة البرمجية الداخلية المكتوبة بلغة التجميع (assembly).

### ب. الأصناف المتشعِّبة

ربما قد صادفت من خلال العمل مع جافا مفهوم إنشاء صنف داخل جسم صنف آخر، أي إنشاء أصناف متشعِّبة؛ ويمكنك فعل الشيء نفسه في كوتلن، ويوضِّح لك المثال التالي كيف يمكنك فعل ذلك:

```
class OuterClassName {
    class NestedClassName {

    }
}
```

يمكنك بالطبع توفير مستوى الوصول للصنف المتشعِّب، وإذا ضبطه إلى خاص `private`، فلن تتمكن من إنشاء كائن من ذلك الصنف `NestedClassName` إلا من داخل نطاق الصنف الخارجي `OuterClassName` فقط.

يجب عليك استخدام الكلمة المفتاحيّة `internal` للسماح لكتلة الشيفرة المصدريّة داخل وحدتك أن تكون قادرة على إنشاء نسخة للأصناف الداخليَّة، وإذا قرَّرت تعيين مستوى الوصول إلى محمي `protected`، يمكن لأي صنف مشتق من الصنف الخارجي `OuterClassName` أن يكون قادرًا على إنشاء هذه النسخ. إذا كان مصطلح الاشتقاق جديد عليك، فلا تقلق، سنتحدث عن الوراثة لاحقًا في هذا الفصل وسيتضح كل شيء لك.

يوجد نوعين من الأصناف المتداخلة في جافا: ساكنة (static) وغير ساكنة (non-static)، وتسمى

الأصناف المتشـعّبة الـتي يصـرّح عنها باسـتخدام الكلمـة المفتاحيّـة static باسـم «الأصـناف المتشـعّبة السـاكنة» (static nested classes)، بينما تسمى الأصناف المُصرّح على أنّها غير ساكنة (non-static) باسـم «الأصـناف الداخليّة» (inner static) ويعدُّ الصنف المتشعّب عضوًا (member) في الصنف الذي يكون ضمنه:

```
class Outer {
      static class StaticNested {}
      class Inner {}
}
```

هنالك فـرق صـغير بين الأصـناف المتشـعّبة السـاكنة والداخليّـة، فيملـك هـذا الأخـير حـق الوصـول إلى أعضـاء الصنف الذي يتضمنها حـتى لـو صُـرّح على أنها خاصّـة private، في حين أنـه يمكن للأصـناف المتشـعّبة السـاكنة الوصول إلى الأعضاء العامة فقط. وعلاوة على ذلك، لإنشـاء نسـخة من الصـنف الـداخلي، فسـتحتاج أولًا إلى نسـخة من الصنف الخارجي Outer.

تدعم كوتلن البناء نفسه الموجود في جافا، فيمكنك استخدام ما يلي لإنشاء ما يعادل صنفًا متداخلًا ثابتًا:

```
class BasicGraph(val name: String) {
   class Line(val x1: Int, val y1: Int, val x2: Int, val y2:Int) {

      fun draw(): Unit {
         println("Drawing Line from ($x1:$y1) to ($x2, $y2)")
      }
   }
   fun draw(): Unit {
      println("Drawing the graph $name")
   }
}
val line = BasicGraph.Line(1, 0, -2, 0)
line.draw()
```

المثـال واضـح ويشـرح نفسـه بنفسـه. فللسـماح للصـنف Line بالوصـول إلى عضـو خـاص private للصـنف الخـارجي BasicGraph، كـل مـا عليـك فعلـه هـو جعـل صـنف Line داخلي، فقـط باسـتخدام الكلمـة المفتاحيـة

:inner

```
class BasicGraphWithInner(graphName: String) {
   private val name: String
   init {
      name = graphName
   }
   inner class InnerLine(val x1: Int, val y1: Int, val x2: Int, val y2:
Int) {
      fun draw(): Unit {
         println("Drawing Line from ($x1:$y1) to ($x2, $y2) for graph $name
")
      }
   }
   fun draw(): Unit {
      println("Drawing the graph $name")
   }
}
```

تأتي كوتلن مع التعبير this القوي الذي قد تكون معتادًا عليه، ويمكنك الإشارة إلى النطاق الخارجي لـ this عن طريق استخدام الباني المسمى this@label وإليك مثال على ذلك:

```
class A {
   private val somefield: Int = 1
   inner class B {
      private val somefield: Int = 1
      fun foo(s: String) {
         println("Field <somefield> from B" + this.somefield)
         println("Field <somefield> from B" + this@B.somefield)
         println("Field <somefield> from A" + this@A.somefield)
      }
   }
}
```

في هذه الحالة، تحتوي كلٌّ من الأصناف الخارجيَّة والداخليَّة على حقل يتقاسم الاسم نفسه، ويساعد التعبير this في التفرقة بينهما.

عند العمل على شيفرة برمجيّة لواجهة المستخدم (UI)، فيجب عليك توفير معالج الحدث (event handler) لعنصر التحكم (زر، مربع القائمة ...إلخ.) لمختلف الأحداث التي ستحدث، والمثال الأكثر شيوعًا هو حدث النقر على زر على شاشتك، إذ ترغب عادةً في التفاعل وتنفيذ بعض الإجراءات. وسيتوقع منك إطار واجهة المستخدم (UI framework) توفير نسخة لصنف؛ وستحتاج إلى الوصول من الصنف المستمع (listener class) هذا إلى حالة ما (state) في نطاق الصنف الخارجي، ولذلك سينتهي بك الأمر إلى توفير صنف داخلي مجهول، كما في المثال التالي الذي يحسب عدد النقرات على الزر:

```
class Controller {
     private var clicks:Int=0
     fun enableHook() {
             button.addMouseListener(object : MouseAdapter() {
                     override fun mouseClicked(e: MouseEvent) {clicks++}
             })
     }
}
```

نفترض أنَّ هنالك مرجع للزر UI وربطنا رد النداء (callback) ـenableHook لأحداث الفأرة؛ فكل مرة يُنقر فيها على الزر، ستزيد قيمة الحقل clicks الذي يمثِّل عدد الضغطات، وكل ما عرّفناه هنا هو في الواقع صنف داخلي واحد مجهول.

## ت. أصناف البيانات

نكون في الكثير من الأحيان بحاجة إلى تعريف الأصناف لغرض واحد وهو احتواء البيانات، فإذا برمجت سابقًا بلغة Scala، أنا متأكد من أنه سيتبادر إلى ذهنك أصناف الحقيبة (case classes). وتوفر كوتلن مفهومًا مماثلًا، لكن يُعرف هذا المصطلح باسم «أصناف البيانات» (data classes) وسنتحدث أكثر قليلًا حول نوع الصنف هذا بالتفصيل في الفصل القادم، لكن في الوقت الحالي، يمكنك تعريف صنف من هذا النوع بالشكل التالي:

```
data class Customer(val id:Int, val name:String, var address:String)
```

يقوم المصرّف بالكثير من الأشياء عندما نعرّف صنف بيانات، لكننا سنترك هذه التفاصيل لوقت لاحق.

## ث. أصناف التعداد

التعداد (enumeration) هو نوع خاص من أنواع الأصناف، فالمتغير من نـوع enum يقتصـر على مجموعـة من الثـوابت المُعرّفـة مسـبقًا الـتي حـددها النـوع. ولتعريـف تعـداد، يجب عليـك اسـتخدام الكلمـة المفتاحيـة enum class، كما في المثال التالي الذي ينشئ نوعًا لجميع أيام الأسبوع:

```
enum class Day {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,SATURDAY,
SUNDAY}
```

يمكن تمرير معاملات إلى بـاني التعـداد مثـل جميـع الأصـناف. فيمكننـا مثلًا تعريـف صـنف enum ليمثِّل جميـع الكــــواكب في مجموعتنــــا الشمســــيّة مــــع الاحتفــــاظ بقيمــــة كــــلٍّ من كتلتــــة الكــــوكب ونصف قطره:

```
public enum class Planet(val mass: Double, val radius: Double) {
MERCURY(3.303e+23, 2.4397e6), VENUS(4.869e+24, 6.0518e6),
EARTH(5.976e+24, 6.37814e6), MARS(6.421e+23, 3.3972e6), JUPITER(1.9e+27,
7.1492e7), SATURN(5.688e+26, 6.0268e7), URANUS(8.686e+25, 2.5559e7),
NEPTUNE(1.024e+26, 2.4746e7);
}
```

أنشأتُ معـاملين من النـوع val ليمثِّلا خاصّـيَّة لكل تعـداد. تـأتي جميـع نسـخ التعـداد مع خاصـيتين مُعرَّفـتين مسـبقًا، الأولى هي name ونوعهـا String (سلسـلة نصـية) والثـاني هي ordinal ونوعهـا int (عـدد صـحيح)، فتمثِّل الأولى اسم النسخة، والأخيرة موضع التعداد بين جميع التعدادات المُعرَّفة.

على غرار جافا، توفر لك كوتلن توابع مساعدة (helper methods) للعمل مـع أصـناف التعـداد. فـإن أردت استرداد قيمة التعداد استنادًا إلى اسمه، ستحتاج إلى استخدام التابع التالي:

```
Planet.valueOf("JUPITER")
```

وللحصول على جميع القيم المُعرَّفة، ستحتاج إلى التابع التالي:

```
Planet.values()
```

مثل أي صنف، يمكن لأنواع التعداد أن ترث واجهة (interface) وتنفذها تنفيذًا مجهولًا لكل قيمة في التعداد، وإليك مثال على كيفيّة تحقيق ذلك:

```
interface Printable {
     fun print(): Unit
}

public enum class Word : Printable {
     HELLO {
            override fun print() {
                   println("Word is HELLO")
            }
     },
     BYE {
            override fun print() {
                   println("Word is BYE")
            }
     }
}
val w= Word.HELLO
w.print()
```

## ج. التوابع الساكنة والكائنات الرفيقة

على عكس جافا، لا تدعم كوتلن التوابع الساكنة (static method) للصنف، ويعرف معظم القراء - الذين لديهم خبرة مسبقة مع جافا - أنَّ التوابع الثابتة لا تنتمي إلى نسخة الكائن بل إلى النوع نفسه. ويُستحسَن في كوتلن تعريف توابع على مستوى الحزمة لتعمل مثل التوابع الثابتة في جافا. لنعرّف ملف كوتلن جديدًا ونسميه `Static`، ونضع داخل هذا الملف شيفرة برمجيّة لدالة سترجع لنا أول حرف من السلسلة النصيّة المعطاة (إذا كانت السلسلة المعطاة فارغة، فسترمي الدالة استثناءً)، كالتالي:

```
fun showFirstCharacter(input:String):Char{
     if(input.isEmpty()) throw IllegalArgumentException()
     return input.first()
}
```

بعـد ذلـك، يمكنـك ببسـاطة اسـتدعاء showFirstCharacter("Kotlin is cool!") في شـيفرتك البرمجيَّة، ينجز المصرّف بعض الأعمـال لـك، ويمكننـا باسـتخدام javap إلقـاء نظـرة على الشـيفرة المولّـدة بصـيغة بايتكود؛ ولفعل ذلك، نفّذ فقط الأمر javap -c StaticKt.class لاستكشاف الشيفرة البرمجيّة التي أنشـأها المُصرِّف:

```
Compiled from "Static.kt"
public final class com.programming.kotlin.chapter03.StaticKt {
   public static final char showFirstCharacter(java.lang.String);
      Code:
         0: aload_0
         1: ldc  #9 //String input
         3: invokestatic #15 //Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/
Object;Ljava/lang/String;)V
         ...
         40: aload_0
         41: checkcast #17 //class java/lang/CharSequence
         44: invokestatic #35 //Method
kotlin/text/StringsKt.first:(Ljava/lang/CharSequence;)C
         47: ireturn
}
```

كمـا تـرى في المخرجـات، أنشـأ المصـرّف صـنفًا لنـا وأعلن على أنَّه final أي ثـابت لا يمكن توريثـه كمـا تعلم. وأضاف المصرّف داخل هذا الصنف الدالة التي عرّفناها. دعنا الآن نستدعي هذا التابع من نقطة دخول البرنامج:

```
fun main(args: Array<String>) {
   println("First lettter:" + showFirstCharacter("Kotlin is cool"))
}
```

ويمكننا مجددًا باستخدام `javap` استكشاف الشيفرة المولدة لنرى ماذا يجري خلف الكواليس:

```
Compiled from "Program.kt"
public final class com.programming.kotlin.chapter03.ProgramKt {
  public static final void main(java.lang.String[]);
  Code:
    0: aload_0
    ...
    18: ldc #29 //String Kotlin is cool
    20: invokestatic #35 //Method
com/programming/kotlin/chapter03/StaticKt.showFirstCharacter:(Ljav a/lang/
String;)C
}
```

استبعدنا معظم بايت كود للتبسيط، لكن في السطر 20، يمكنك أن ترى أن هنالك استدعاء لتابعنا، وبالضبط، تم الاستدعاء عن طريق البرنامج `invokestatic`.

لا يمكننا التحدث عن التوابع الساكنة دون ذكر نمط المفردة (singleton) وهو نمطٌ من أنماط التصميم (design pattern) يحد من إنشاء نسخ لصنف معين إلى نسخة واحد؛ وبمجرّد إنشاء تلك النسخة المفردة، ستعيش طوال فترة البرنامج. تستعير كوتلن النهج الموجود في لغة سكالا، وهكذا يمكنك تعريف كائن مفرد في كوتلن الشكل التالي:

```
object Singleton{
  private var count = 0
  fun doSomething():Unit {
    println("Calling a doSomething (${++count} call/-s in total)")
  }
}
```

يمكنك الآن استدعاء `Singleton.doSomething` من أي دالة، وسيزداد العداد `count` في كل مرة آنذاك، وإذا نظرت إلى البايتكود المتولد، فستكتشف أنَّ المصرّف ينجز بعض الأعمال لنا مرةً أخرى:

```
public final class com.programming.kotlin.chapter03.Singleton {
```

```
    public static final com.programming.kotlin.chapter03.Singleton
    public final void doSomething();
    Code:
      0: new #10 // class java/lang/StringBuilder
      43: return
      ...
    static {};
    Code:
      0: new #2 //class com/programming/kotlin/chapter03/Singleton
      3: invokespecial #61 //Method "<init>":()V
      6: return
  }
```

لقد استبعدتُ الشيفرة البرمجية المولدة للتابع `doSomething` الخاصة بنا نظرًا لأنها ليست محط اهتمامنا.

لقد أنشئ المصرّف مرةً أخرى صنفًا وأعلن على أنه `final`؛ وعلاوةً على ذلك، أضاف عضوًا يسمى `INSTANCE` وأعلن على أنه `static`، والجزء المثير للاهتمام هو في نهاية القائمة حيث سترى نقطة الدخول `static{};` وهذا هو مُهيِّئ الصنف، ويُستدعَى مرةً واحدةً، وسيتأكد JVM من حدوث ذلك قبل:

- إنشاء نسخة للصنف.
- استدعاء تابع ساكن للصنف.
- إسناد قيمة لحقل ساكن للصنف.
- استخدام حقل ساكن غير ثابت (non-constant).
- تنفيذ تعليمة توكيد (assert statement) متداخلة ضمن الصنف لصنف ذا مستوى أعلى (top-level class).

تُستدعى الشيفرة البرمجيّة في هذه الحالة قبل أول استدعاء لـ `doSomething` لأننا نصل إلى العضو الثابت `INSTANCE` (انظر إلى البرنامج `getstatic` في شيفرة البايتكود التالية)، وإذا استدعينا هذا التابع مرتيّن، سنحصل على شيفرة البايتكود التالية:

```
public static final void main(java.lang.String[]);
   Code:
      0: aload_0
      1: ldc #9 // String args
      3: invokestatic #15 //Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/
Object;Ljava/lang/String;)V
      6: getstatic #21 //Field
com/programming/kotlin/chapter03/Singleton.INSTANCE:Lcom/programming/
kotlin/chapter03/Singleton;
      9: invokevirtual #25 //Method
com/programming/kotlin/chapter03/Singleton.doSomething:()V
      12: getstatic #21 //Field
com/programming/kotlin/chapter03/Singleton.INSTANCE:Lcom/programming/
kotlin/chapter03/Singleton;
      15: invokevirtual #25 //Method
com/programming/kotlin/chapter03/Singleton.doSomething:()V
      18: return
```

يمكنك أن ترى أنه في كلتا المناسبتيّن، يُستدعَى doSomething على أنَّه تابع افتراضي، والسبب هو أنه يمكنك إنشاء صنف مفرد يرث من صنف معين، كما في المثال التالي:

```
open class SingletonParent(var x:Int){
      fun something():Unit{
             println("X=$x")
      }
}
object SingletonDerive:SingletonParent(10){}
```

هنالك طريقة لاستدعاء تابع ساكن كما في جافا؛ ولفعل ذلك، ستحتاج إلى وضع كائنك داخل الصنف وتحديده على أنه كائن رفيق (companion object). لا شك أنَّ مفهوم الكائن الرفيق مألوفٌ لمن لديه معرفة أساسيّة بلغة سكالا، يستخدم المثال التالي نمط تصميم المصنع (factory design pattern) لإنشاء نسخة من الصنف Student:

```
interface StudentFactory {
      fun create(name: String): Student
}
class Student private constructor(val name: String) {
      companion object : StudentFactory {
             override fun create(name: String): Student {
                    return Student(name)
             }
      }
}
```

كما ترى، حُدد باني النوع Student على أنَّه خاص private، وبالتالي، لا يمكن استدعاؤه من أي مكان سوى من داخل الصنف Student أو من companion object (كائن رفيق). يملك الصنف companion رؤية كاملة لجميع توابع وأعضاء الصنف Student.

من الشيفرة البرمجية، ستحتاج إلى استدعاء Student.create("Jack Wallace") لإنشاء نسخة جديد من الصنف Student. إذا نظرّت في مخرجات البناء، ستلاحظ وجود صنفيّن مولَّدين للصنف Student: الأول هو Student.class والثاني هو Student$Companion.class. لنرَ كيف يترجم استدعاء Student.create في بايتكود:

```
public final class com.programming.kotlin.chapter03.ProgramKt {
   public static final void main(java.lang.String[]);
   Code:
      0: aload_0
      1: ldc #9 //String args
      3: invokestatic #15 //Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/
Object;Ljava/lang/String;)V
      6: getstatic #21 // Field
com/programming/kotlin/chapter03/Student.Companion:Lcom/programming/
kotlin/chapter03/Student$Companion;
      9: ldc #23 //String Jack Wallace
      11: invokevirtual #29 //Method
```

```
com/programming/kotlin/chapter03/Student$Companion.create:(Ljava/lang/
String;)Lcom/programming/kotlin/chapter03/Student;
      14: pop
      15: return
}
```

ستلاحظ في السطر 6 أنَّ هنالك استدعاء للعضو الثابت `getstatic`؛ وكما تتخيل، أضيف حقل ثابت إلى الصنف `Student` من النوع `Student.Companion`:

```
public final class com.programming.kotlin.chapter03.Student {
   public static final com.programming.kotlin.chapter03.Student$Companion
Companion;
   public final java.lang.String getName();
      static {};
      Code:
         0: new #39 //class
com/programming/kotlin/chapter03/Student$Companion
         3: dup
         4: aconst_null
         5: invokespecial #42 //Method
com/programming/kotlin/chapter03/Student$Companion."<init>":(Lkotlin/jvm/
internal/DefaultConstructorMarker;)V
         8: putstatic #44 //Field
Companion:Lcom/programming/kotlin/chapter03/Student$Companion;
         11: return
public
com.programming.kotlin.chapter03.Student(java.lang.String,kotlin.jvm.inter
nal.DefaultConstructorMarker);
Code:
   0: aload_0
   1: aload_1
   2: invokespecial #24 //Method "<init>":(Ljava/lang/String;)V
   5: return
```

يثبت هذا الجزء من الشيفرة أنَّ الافتراض الذي افترضناه صحيح، ويمكنك رؤية كيف أضيف العضو

Companion إلى صنفنا. ومرة أخرى، يحصل الصنف على شيفرة باني صنف مولّد لإنشاء نسخة للصنف الرفيق، وأنّ Student.create هو اختصار لكتابة شيفرة برمجية مثل ()Student.Companion.create. وإذا كنت تحاول إنشاء نسخة من Student.Companion (أي val c = Sudent.Companion)، فستحصل على خطأ في التصريف. فيتبع الكائن الرفيق جميع قواعد الوراثة.

## 2. الواجهات

الواجهة (interface) هي مجرّد عقد (contract) تحتوي على تعريفات لمجموعة وظائف ذات صلة. ويجب أن يلتزم مُنفّذ الواجهة بعقّد الواجهة وتنفيذ التوابع المطلوبة. بشكل مشابه للإصدار 8 من جافا، تحتوي واجهة كوتلن على تصريحات التوابع المجرّدة وكذلك تنفيذات التابع؛ وعلى عكس الأصناف المجرّدة، لا يمكن للواجهة أن تحتوي على حالة (state)، ويمكن، مع ذلك، أن تحتوي على خاصيات. سيجد مطوري Scala أنّ هذا مشابه لسمات لغة سكالا (أي Scala traits):

```
interface Document {
     val version: Long
     val size: Long

     val name: String
     get() = "NoName"

     fun save(input: InputStream)
     fun load(stream: OutputStream)
     fun getDescription(): String {
           return "Document $name has $size byte(-s)"}
     }
```

تعرّف الواجهة ثلاثة خاصيات وثلاثة توابع؛ توفر الخاصيّة name والتابع getDescription التنفيذ الافتراضي، لكن كيف نستخدم الواجهة من صنف جافا؟ دعنا نرى من خلال تنفيذ هذه الواجهة:

```
public class MyDocument implements Document {
```

```
        public long getVersion() {
            return 0;
        }

        public long getSize() {
            return 0;
        }

        public void save(@NotNull InputStream input) {
        }

        public void load(@NotNull OutputStream stream) {
        }

        public String getName() {
            return null;
        }

        public String getDescription() {
            return null;
        }
    }
```

تستطيع أن ترى أنَّ الخاصيات تُرجمَت إلى جالبات (Getters). رغم أنَّك توفِّر التنفيذ الافتراضي للتابع `getDescription` مع الاسم إلا أنه لا يزال عليك تنفيذهما، وهذا ليس أمرًا مهمًا عند تنفيذ واجهة في صنف كوتلن:

```
class DocumentImpl : Document {
    override val size: Long
    get() = 0

    override fun load(stream: OutputStream) {
```

```
    }

    override fun save(input: InputStream) {
    }
    override val version: Long
    get() = 0
  }
```

دعنا نرى ما الذي يحدث خلف الكواليس مع الشيفرة البرمجيّة لهذين التابعين المنفذين على مستوى الواجهة:

```
$ javap -c build\classes\main\com\programming\kotlin\chapter03\
DocumentImpl.class
Compiled from "KDocumentImpl.kt"
public final class com.programming.kotlin.chapter03.KDocumentImpl
implements com.programming.kotlin.chapter03.Document {
  public long getSize();
    Code:
      0: lconst_0
      1: lreturn
public void load(java.io.OutputStream);
  Code:
    0: aload_1
    1: ldc           #15   //String stream
    3: invokestatic  #21   //Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/Ob
ject;Ljava/lang/String;)V
    6: return
public void save(java.io.InputStream);
  Code:
    0: aload_1
    1: ldc           #26    //String input
    3: invokestatic  #21   //Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/Ob
```

```
ject;Ljava/lang/String;)V
     6: return
public long getVersion();
    Code:
      0: lconst_0
      1: lreturn
public com.programming.kotlin.chapter03.KDocumentImpl();
    Code:
      0: aload_0
      1: invokespecial #32 //Method  java/lang/Object."<init>":()V
      4: return
public java.lang.String getName();
   Code:
     0: aload_0
     1: invokestatic  #39  //Method
com/programming/kotlin/chapter03/Document$DefaultImpls.getName:(Lcom/
programming/kotlin/chapter03/Document;)Ljava/lang/String;
   4: areturn
public java.lang.String getDescription();
   Code:
     0: aload_0
     1: invokestatic  #43 //Method
com/programming/kotlin/chapter03/Document$DefaultImpls.getDescription :
(Lcom/programming/kotlin/chapter03/Document;)Ljava/lang/String;
     4: areturn
}
```

ربما لاحظت الاستدعاءات إلى الصنف `DefaultImpls` في شيفرة `getDescription` و `getName`، وإذا نظرت إلى الأصناف التي ولدها المُصرِّف (`build/main/com/ programming/kotlin/chapter03`) ستلاحظ ملفًا باسم `Document$DocumentImpls.class`.

أسمعك تسأل، ما هذا الصنف الذي لم نكتبه؟ يمكننا معرفة ما يحتويه عن طريق `javap`:

```
public final class com.programming.kotlin.chapter03.Document$DefaultImpls
{
   public static java.lang.String
getName(com.programming.kotlin.chapter03.Document);
     Code:
       0: ldc            #9                  //String NoName
       2: areturn
public static java.lang.String
getDescription(com.programming.kotlin.chapter03.Document);
     Code:
       0: new            #14        //class java/lang/StringBuilder
       3: dup
       4: invokespecial #18                //Method
java/lang/StringBuilder."<init>":()V
       7: ldc            #20           //String Document
       9: invokevirtual #24               //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilde
r;
      12: aload_0
       13: invokeinterface #29,  1         //InterfaceMethod
com/programming/kotlin/chapter03/Document.getName:()Ljava/lang/String ;
       40: invokevirtual #43            //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
       43: areturn
}
```

من جزء الشيفرة البرمجيّة السابق (استبعدّت بعض الأجزاء لتبسيط الشرح)، يمكنك أن ترى بوضوح أنّ المصرّف قد أنشأ صنفًا لنا يحتوي على تابعيّن ساكنين يتطابق مع اللذين وفرّنا تنفيذًا لهما في الواجهة.

لمّا كانت الشيفرة البرمجيّة للتابع getName بسيطة للغاية (إعادة سلسلة نصيّة)، فإن الشيفرة البرمجيّة للتابع getDescription معقّدة أكثر، فهي تستخدم StringBuilder لإنشاء سلسلة نصيّة لأغراض الوصف. والجزء المثير للاهتمام هو المتعلق بالتابعين getSize و getName؛ فلو نظرّت إلى السطر 12، ستجد أنّ aload_0 تدفع بمعامل Document (يأخذ التابع getDescription معاملًا واحدًا) إلى المكدّس،

ويستدعي السطر السابق عبر استخدام `invokeinterface` لاستدعاء تابع مُعرَّف بوساطة واجهة جافا.

إنَّ مناقشة تفاصيل شيفرة بايتكود لجافا يتجاوز نطاق هذا الكتاب، ويمكنك العثور على تفاصيل أوسع - إذا كنت تريد الاستزادة – بإجراء بحث سريع على الإنترنت.

## 3. الوراثة

الوراثة هي جزء أساسي من البرمجة الكائنيّة التوجّه، فهي تسمح لنا بإنشاء أصناف جديدة قابلة لإعادة الاستخدام، و/أو التوسعة، و/أو تعديل سلوك تلك الموجودة مسبقً؛ وتدعى الأصناف الموجودة مسبقًا «بالأصناف العليا» (super class، أو الأساس [base] أو الأبوية [parent])، وتسمى الأصناف الجديدة الناتجة «بالأصناف المشتقّة» (derived class)، وتوجد بعض القيود حول عدد الأصناف العليا التي يمككنا الوراثة منها؛ ففي JVM، تستطيع أن ترث من صنف أساسي (أب) واحد لكنك تستطيع أن ترث من واجهات متعدّدة.

إنَّ عملية الوراثة متوارثة كتوارث الآباء أملاك الأجداد وتوارث الأبناء بالمثل أملاك أولئك الآباء وهلم جرًّا؛ فإذا اشتققت الصنف C من الصنف B الذي هو مشتق من الصنف A، فإن الصنف C يعدُّ مشتقًّا من الصنف A ويرثه.

سيرث الصنف المشتق ضمنيًّا من جميع الأصناف الأبوية (وآباء تلك الأصناف الأبوية وهكذا دواليك) وهذا يتضمن الحقول والخاصيات والتوابع. وتكمن أهميّة الوراثة في القدرة على إعادة استخدام الشيفرة التي كُتبَت مسبقًا وبالتالي تجنُّب الحالة التي نضطر فيه إلى إعادة تنفيذ السلوك الموجود في الصنف الأب.

يمكن للصنف المُشتَق إضافة حقول أو خاصيات أو توابع جديدة خاصة به وبالتي يُوسّع من الوظائف المتاحة في الصنف الأب. ويمكن أن نقول أنَّ الصنف المشتق B يُخصِّص الصنف A الذي هو الصنف الأب. وأبسط مثال يمكنك أن تفكِّر فيه ويوضح لك الفكرة هو المخطط البياني للمملكة الحيوانيَّة، إذ في الأعلى ستجد الحيوانات وتليها الفقاريات واللافقريات، وينقسم الصنف الأول إلى الأسماك والزواحف والثدييات ...إلخ. فإذا أخذنا أنواع التونة ذات الزعنفة الصفراء، فيمكننا النظر إليها كنوع مخصص من الأسماك.

يوضح الرسم التوضيحي التالي تسلسلًا هرميًا بسيطًا للصنف؛ لنفترض أنك كتبت نظامًا للتعامل مع المدفوعات، سيكون لديك صنفًا يسمى `Payment` يحتوي على المبلغ وصنفًا آخر يدعى `CardPayment` يتولى المدفوعات:

| Any |
| --- |
| + equals( arg list ) : Boolean<br>+ hashCode() : Int<br>+ toString( ) : String |

| Payment |
| --- |
| - amount:Decimal |
| + equals( arg list ) : Boolean<br>+ hashCode() : Int<br>+ toString( ) : String<br>+ getAmount():Decimal |

| CardPayment |
| --- |
| - amount:Decimal<br>- number:String<br>- expiryDate: DateTime<br>- type:CardType |
| + equals( arg list ) : Boolean<br>+ hashCode() : Int<br>+ toString( ) : String<br>+ getAmount() : Decimal<br>+ getNumber() : String<br>+ getExpiryDate() : DateTime<br>+ getType(): CardType |

ستلاحظ وجود كيان آخر يسمى Any في الصورة السابقة، ففي كل مرة تنشئ كيانًا لا يرث من أي كيان (صنف) آخر، فسيرث آنذاك من هذا الكيان ويَعُدُّه أبًا له. ربما ستعتقد أنَّ Any يماثل الصنف Object، الذي هو الصنف الأصل لأي صنف معرّف في جافا، لكن هذا ليس صحيحًا. فلو انتبهت إلى التوابع المعرّفة في الصنف Any، ستلاحظ أنها مجموعة فرعيّة من تلك الموجودة في صنف Object جافا، فكيّف تتعامل كوتلن مع المراجع التي تشير إلى الكائن Object في جافا؟ عندما يرى المُصرّف مثل هذا الكائن، فسيُحوِّله إلى الصنف Any وبالتالي سيتمكن من استخدام التوابع الملحقة لاستكمال مجموع التابع.

لنطبّق الشيفرة البرمجية السابقة ونرى كيف تعمل الوراثة وتُعرَّف في كوتلن:

```
enum class CardType {
      VISA, MASTERCARD, AMEX
    }

    open class Payment(val amount: BigDecimal)
    class CardPayment(amount: BigDecimal, val number: String, val
expiryDate: DateTime, val type: CardType) : Payment(amount)
```

أنشأنا أصنافا بناءً على المواصفات التي رأيناها للتو. فكما هو موضّح في التعريف، CardType من نوع تعداد (enumeration)، ولقد قدّم لنا التعريف كلمة مفتاحية جديدة هي open. فعبر هذه الكلمة المفتاحية، تعلن على أن هذا الصنف يمكن الوراثة منه، فلقد قرر مطورو كوتلن أنَّ الأصناف ستكون غير قابلة للوراثة افتراضيًا. فإذا

برمجت مسبقًا باستخدام جافا، فلقد صادفت الكلمة المفتاحية `final`، والتي هي عكس `open`؛ ففي جافا، يمكن الوراثة من أي صنف إذا لم تضع `final` عليه. ويعلن تعريف الصنف `CardPayment` أنّ الوراثة تكون عبر النقطتين، إذ معنى `: Payment` إلى "الصنف `CardPayment` الموسّع من الصنف `Payment`"، ويختلف هذا عن جافا التي تجري عملية الوراثة فيها عبر استخدام الكلمة المفتاحية `extends`، وسيعرف أي مطوّر سي بلس بلس أو سي شارب هذا البناء.

في الشيفرة البرمجيّة السابقة، يملك الصنف CardPayment بانيًا أساسيًا، وبالتالي، يجب استدعاء الأصل `Payment(amount)`، لكن ماذا لو لم يُعرِّف صنفنا الجديد بانيًا أساسيًا؟ لنوسّع التسلسل الهرمي للصنف ونضف صنفًا جديدًا باسم `ChequePayment`:

```
class ChequePayment : Payment {
        constructor(amount: BigDecimal, name: String, bankId: String) :
super(amount) {
            this.name = name
            this.bankId = bankId
        }
        var name: String
            get() = this.name
        var bankId: String
            get()  = this.bankId
    }
```

بما أننا اخترنا تجنّب الباني الأساسي، فيجب على تعريف الباني الثانوي استدعاء الباني الموروث من الأصل، ويجب أن يكون هذا الاستدعاء هو أول شيء يقوم به الباني؛ وبالتالي، يسبق `super(args1,args2...)` جسم الباني، ويختلف هذا عن جافا، حيث أننا ننقل هذا الاستدعاء إلى السطر الأول من جسم الباني.

في هذا المثال، نحن نرث من صنف واحد فقط، فلقد قلنا سابقًا أننا لا يمكن أن نرث من أكثر من صنف واحد، ومع ذلك، يمكننا أن نرث من واجهات متعدّدة في الوقت نفسه. ولنأخذ مثالًا بسيطًا على سيارة برمائيّة: فهي قارب وسيارة، وإذا أردت وضع نموذج لها، فسنعد أن لها واجهتين: `Drivable` و `Sailable`، وسترث (تتوسع) سيارتنا البرمائية منهما على النحو التالي:

```
interface Drivable {
        fun drive()
    }
    interface Sailable {
        fun saill()
    }
    class AmphibiousCar(val name: String) : Drivable, Sailable {
        override fun drive() {
            println("Driving...")
        }
        override fun saill() {
            println("Sailling...")
        }
}
```

تذكّر أنَّ الصنف يُشتق تلقائيًا من Any، أي يبدو الأمر كما لو كتبنا:

```
class AmphibiousCar(val name:String):Any, Drivable, Sailable.
```

عندما نرث من واجهة، يجب علينا توفير تنفيذ لجميع التوابع والخاصيات أو يجب علينا أن نجعل الصنف مجرّدًا، وسنتحدث قريبًا عن الأصناف المجرّدة. لا يوجد تقييد على عدد الواجهات التي يمكنك الوراثة منها والترتيب الذي تُحدده. خلافًا لجافا، إذا ورث صنف من صنفٍ وواجهةٍ واحدةٍ أو أكثر، فلست مقيدًا بإدراج اسم الصنف أولًا في قائمة الأصول الموروث منها ثمَّ إدراج أسماء الواجهات بل يمكن وضع اسم واجهةٍ ما أولًا على الشكل التالي:

```
interface IPersistable {
        fun save(stream: InputStream)
    }

    interface IPrintable {
        fun print()
```

```
    }

    abstract class Document(val title: String)

    class TextDocument(title: String) : IPersistable, Document(title),
IPrintable {
  override fun save(stream: InputStream) {
          println("Saving to input stream")
       }

       override fun print() {
          println("Document name:$title")
       }
    }
```

## 4. رؤية المغيرات

يمكن أن تمتلك الأصناف أو التوابع أو الخاصيات أو الحقول عند تعريفها على مستويات مرئية مختلفة، وتوجد في كوتلن أربعة قيم محتملة:

- عام public: يمكن الوصول إلى الصنف من أي مكان.
- داخلي internal: يمكن الوصول إليه من الشيفرة البرمجية للوحدة.
- محمي protected: يمكن الوصول إليه فقط من الصنف الذي يُعرّفه وأي أصناف مشتقة.
- خاص private: يمكن الوصول إليه من نطاق الصنف الذي يُعرّفه.

إذا حَدَّد الصنف الأصل أنَّ حقلا معينًا متاحٌ لإعادة تعريفه (أي استبداله [overwritten]) عبر الكلمة المفتاحية open، فإنَّ الصنف المشتق سيكون قادرًا على تعديل مستوى المرئية، وإليك مثال على ذلك:

```
open class Container {
      protected open val fieldA: String = "Some value"
}
```

```
class DerivedContainer : Container() {
   public override val fieldA: String = "Something else"
}
```

الآن في الصنف الرئيسي، يمكنك إنشاء نسخة DerivedContainer وطباعة قيمة الخاصية fieldA، إذ أصبح هذا الحقل عامًّا لأي شيفرة برمجيّة:

```
val derivedContainer =  DerivedContainer()
     println("DerivedContainer.fieldA:${derivedContainer.fieldA}")
    /*val container:Container = derivedContainer
println("fieldA:${container.fieldA}")*/
```

لقد علّقت الشيفرة البرمجية التي استخدمنا فيها derivedContainer كأنَّها نسخةٌ من DerivedContainer؛ ففي هذه الحالة، عند محاولة تصريف الشيفرة البرمجية الموجودة المُعلّقة، فسينتج خطأ لأنَّه لا يمكن الوصول إلى الخاصية fieldA.

لا يعني إعادة تعريف الحقل أنَّه سيحل محل الموجود عندما يتعلق الأمر بالكائن المحجوز في الذاكرة (object allocation). تذكَّر أنَّ الصنف المشتق سيرث جميع حقول الصنف الأصل، ويتطلب الأمر بعض الشيفرات البرمجية لإثبات هذا:

```
derivedContainer.javaClass.superclass.getDeclaredFields().forEach  {
        field->
        field.setAccessible(true)
        println("Field:${field.name},${Modifier.toString(field.modifiers)}
, Value=${field.get(derivedContainer)}")
     }
derivedContainer.javaClass.getDeclaredFields().forEach {
   field->
   field.setAccessible(true)
   println("Field:${field.name},${Modifier.toString(field.modifiers)} ,
Value=${field.get(derivedContainer)}")
}
```

ستطبع الشيفرة البرمجيّة السابقة الخاصية `fieldA` مرتيَن في مجرى الخرج عند تشغيلها، وسيكون الخرج الأول من الصنف الأصل وهو "Some Value" والثاني من الصنف المشتق وهو "Something else".

الاستخدام النموذجي لهذا سيكون توسيع الوصول إلى حقل و/أو تابع و/أو خاصيّة معيّنة، ولكن يجب أن تحذر من استخدام هذا الأمر نظرًا لأنه قد يكسر «مبدأ ليسكوف للاستبدال» (Liskov substitution principle)، وباتباع هذا المبدأ، إذا كان البرنامج يستخدم صنفًا أساسيًا، فإنه يمكن استبدال الإشارة إلى هذا الصنف الأساسي مع صنف مشتق دون التأثير على وظيفة البرنامج.

## 5. الأصناف المجرَّدة

عند وضع الكلمة المفتاحية `abstract` أمام تعريف الصنف، سيميّز على أنّه مُجرَّد (abstract)، والصنف المجرّد هو صنف معرّف جزئيًا، ويجب أن يوفر الصنف الوارث تنفيذًا للخاصيات والتوابع التي لا يوفر الصنف المجرَّد الموروث تنفيذًا لها، ما لم تقرر أن يكون ذاك الصنف الوارث (المشتق) هو صنف مجرّد أيضًا. وهذه هي طريقة تعريف الصنف المجرّد في كوتلن:

```
abstract class A {
        abstract fun doSomething()
}
```

يجب عليك إضافة الكلمة المفتاحية `abstract` إلى الدالة إذا لم توفّر لها جسمًا (تنفيذًا) وهذا على عكس الواجهات.

لا يمكنك إنشاء نسخة من صنف مجرّد، فدور هذا الصنف هو توفير مجموعة واحدة من التوابع تشترك بها جميع الأصناف الوارثة منه. وخير مثال على هذا هو الصنف `InputStream`، وسيكون هذا مألوفًا لمطوري جافا. فيقول توثيق JDK: «هذا الصنف المجرّد هوّ الصنف الأعلى لجميع الأصناف التي تمثّل مجرى دخل (input stream) من البايتات. فيجب على التطبيقات التي تحتاج إلى تعريف صنف فرعي من `InputStream` توفير تابع دائمًا يُرجع البايت التالي من المدخلات». وإذا نظرّت إلى الحزمة `java.io`، فستجد بعض التنفيذات لها:

AudioInputStream و ByteArrayInputStream و FileInputStream وغيرها، ويمكنك أنت أيضًا توفير تنفيذ لها.

يمكنك وراثة الصنف A مع دالة أشير إلى أنَّها متاحة عبر الكلمة المفتاحية open وذلك لإعادة تعريفها (قابلة للاستبدال [overridable] كما سنرى قريبًا) وجعلها مجرَّدة عبر الكلمة المفتاحية abstract في الصنف المشتق. ويمكن بهذه الطريقة أن يصبح الصنف المشتق مجردًا، وسيحتاج أي صنف يرث منه إلى توفير التنفيذ، ولن يتمكن من الوصول إلى التنفيذ المعرّف في الصنف A:

```
open class AParent protected constructor() {
   open fun someMethod(): Int = Random().nextInt()
}
abstract class DDerived : AParent() {
   abstract override fun someMethod(): Int
}
class AlwaysOne : DDerived() {
   override fun someMethod(): Int {
      return 1
   }
}
```

إنَّ المثال واضح للغاية، فلدينا صنف أصل يعرّف التابع someMethod الذي يعيد عددًا صحيحًا عشوائيًّا. يرث الصنف Dderived من هذا الصنف (لاحظ أنه يجب علينا استدعاء باني فارغ في الصنف الأصل) ويعلن أنَّ ذلك أصبح مجرّدًا، ومن ثم يوفر الصنف AlwaysOne - الذي يرث من الصنف Dderived - تنفيذًا يعيد دومًا القيمة 1 لذاك التابع.

## 6. واجهة أم صنف مجرَّد؟

هناك دائما نقاش حول استخدام إمّا واجهة أو صنف مجرّد، وهذه بعض القواعد التي يجب اتباعها عند اتخاذ قرار بشأن الطريقة التي يجب اتباعها:

- «هل هو ...» (Is-a) مقابل «هل يمكنه فعل ...» (Can-Do): يمكن لأي نوع أن يرث من صنف أصل واحد

وعدة واجهات، لذا إن لم يكن بإمكانك القول أنَّ الصنف `B` المشتق هو `A` (أي أنَّ `A` هو النوع الأصل)، فلا تستخدم واجهة في هذه الحالة بل استعمل صنفًا، فالواجهات تطبّق علاقة Can-Do. وإذا كانت وظيفة العلاقة Can-Do قابلة للتطبيق على أنواع كائنات مختلفة، فاستخدم تنفيذ واجهة؛ فعلى سبيل المثال، يمكن أن تقول أن كل من `FileOutputStream` و `ByteOutputpuStream` (وأي من التنفيذات المشابهة المتاحة) تملك علاقة Is-a مع `java.io.OutputStream`، وبالتالي سترى أنَّ `OutputStream` هو صنف مجرّد يوفر تنفيذ شائع لجميع الكائنات التي تمثّل مجرى قابل للكتابة. ومع ذلك، فإنَّ `Autocloseable`، الذي يمثّل كائنًا يُمسِك بموردٍ يمكن أن يُترَك عند استدعاء التابع `close`، يوفّر وظيفة Can-do وبالتالي من المنطقي أن يكون ذلك واجهة.

- تعزيز إعادة استخدام الشيفرة البرمجيّة: أنا متأكد من أنك توافق على أنه من السهل وراثة صنف عوضًا عن واجهة يتعيّن عليك فيها توفير تنفيذ لجميع التوابع المعرّفة. فيمكن للصنف الأصل توفير مجموعة كبيرة من الوظائف المشتركة، وبالتالي يمكن للصنف المشتق إمّا إعادة التعريف أو تنفيذ مجموعة من التوابع المُعرَّفة فقط.
- الإصدار: إذا كنت تعمل مع واجهة وأضفت عضوًا جديدًا إليها، فيمكنك إجبار جميع الأصناف المشتقّة على تغيير شيفرتها البرمجيّة عن طريق إضافة تنفيذ جديد لذلك العضو، وتتغيّر آنذاك الشيفرة المصدرية ويتعيَّن إعادة تصريفها من جديد. ولا ينطبق الشيء نفسه على الصنف المجرّد، إذ يمكنك إضافة تابع جديد والاستفادة منه، ولا تحتاج إلى إعادة تصريف الشيفرة البرمجية الخاصة بالمستخدم.

## 7. التعددية الشكلية

بعد التغليف والوراثة، يُنظَر إلى التعددية الشكلية (polymorphism) على أنَّها الركيزة الثالثة للبرمجة كائنيَّة التوجه، فهي تفصل «ماذا» عن «كيف» على مستوى النوع، وواحدة من المزايا التي تقدمها التعددية الشكلية هي تحسين تنظيم الشيفرات المصدرية وتسهيل قراءتها؛ وعلاوةً على ذلك، فإنَّه يُسمَح بتوسيع برامجك في أي وقت لاحقًا، مثل أن ترغب بإضافة مميزات جديدة.

يأتي أصل الكلمة الأجنبية `polymorphism` (معناها «التعددية الشكلية») من اللغة اليونانيّة؛ فالشطر الأول polys (πολύς) يعني «الكثير» (many أو much) والشطر الثاني morphē (μορφή) يعني «نموذج» (

form) أو «شكل» (shape). توجد أنواع عديدة من التعددية الشكلية، لكن سنتحدث في هذا الفصل فقط على «الربط المتأخر» (late- binding، أو «الربط الحيّوي» [dynamic binding] أو «الربط وقت التشغيل» [ [runtime binding]).

سترى قوّة التعددية الشكلية وقت التشغيل عندما تُعامَل كائنات صنف مشتق معاملة كائنات صنف أب، ويمكن أن يحدث هذا لمعامل التابع (method parameter) أو عندما يتعلّق الأمر بتخزين مجموعة من العناصر المشتركة في مجموعة أو مصفوفة؛ والشيء الغريب هنا هو أنه لن يكون النوع المصرح به للكائن متطابق مع النوع الفعلي وقت التشغيل في أثناء تنفيذ الشيفرة، إذ يبدو هذا وكأنه سحر، لكن يحدث كل هذا من خلال استخدام التوابع الافتراضيّة (virtual method).

يمكن للأصناف الأساس (base classes) تعريف توابع افتراضيّة وتنفيذها ويمكن للأصناف المشتقّة استبدالها وتوفير تنفيذ خاص بها. بهذه الطريقة، يتصرف نوعان مختلفان بشكل مختلف عند استدعاء التابع نفسه. عندما يُستدعَى تابع افتراضي أثناء عمل برنامجك، تبحث آلة جافا الافتراضية JVM عن نوع النسخة وقت التشغيل لاكتشاف أي تابع ينبغي استدعاؤه. سنخصص وقت لاحق من هذا الفصل بعض المساحة لمناقشة هذا الأمر بتفصيل أوسع حول كيفيّة تطبيقها.

تُوحِّد التوابع الافتراضيّة كيفية العمل مع مجموعة من الأنواع ذات الصلة. تخيّل العمل على تطبيق رسم كبير، ويجب أن يدعم تصيير (rendering) مجموعة كبيرة من الأشكال على الشاشة، فيجب على البرنامج آنذاك أن يتتبع جميع الأشكال التي سينشئها المستخدم ويتفاعل مع مدخلاته: تغيير موقعها على الشاشة، تغيير خاصياتها (لون الحدود أو الحجم أو لون الخلفيّة ...إلخ)، وعند تصريف الشيفرة البرمجيّة، لا يمكنك أن تعرف مسبقًا جميع أنواع الأشكال التي ستدعمها، وآخر شيء سترغب في القيام به هو التعامل مع كل واحد على حدة.

ستساعدك التعددية الشكلية في مثل هذه الحالات، فأنت تريد معالجة كافة النسخ الرسوميّة كشكل، عن طريق تفاعل الصورة لنقرة المستخدم على لوحة الرسم (canvas) وستحتاج شيفرتك البرمجيّة إلى العمل إذا كان موقع الفأرة داخل حدود أحد الأشكال المرسومة، وما يجب عليك تجنبه هو الدوران حول جميع الأشكال، واستدعاء تابع مختلف للتحقق من حدث ضغطة الفأرة: استدعاء `isWithinCircle` لشكل الدائرة و `checkIsHit` للشكل المعيّن وهلم جرًّا.

لنرَ الآن كيف يمكننا تنفيذ ذلك باستخدام نهج الكتاب. سنُعرّف أولا الصنف Shape، ويجب أن يكون هذا صنفًا مجردًا ولا يمكن إنشاء نسخة منه، لكن كيف يمكن رسم الشكل على الشاشة ما لم يُحدَّد هذا الشكل؟ أرجو أن تلقِ نظرة على الشيفرة البرمجيّة التالية:

```
abstract class Shape protected constructor() {
   var XLocation: Int
      get() = this.XLocation
      set(value: Int) {
         this.XLocation = value
      }

   var YLocation: Int
      get() = this.XLocation
      set(value: Int) {
         this.XLocation = value
      }
   var Width: Double
      get() = this.Width
      set(value: Double) {
         this.Width = value
      }
   var Height: Double
      get() = this.Height
      set(value: Double) {
         this.Height = value
      }
   abstract fun isHit(x: Int, y: Int): Boolean
}
```

الآن ومع هذه الشيفرة، سنُعرِّف تنفيذًا لنوعين من الأشكال: شكل بيضوي (الصنف Ellipsis) وشكل مستطيل (الصنف Rectangle)، وسؤالي لك: هل من المعقول إنشاء شكل مربّع؟ فكّر في هذا، والآن دعنا نكتب

تنفيذ هذين الشكلين آنفي الذكر:

```
class Ellipsis : Shape() {
   override fun isHit(x: Int, y: Int): Boolean {
      val xRadius = Width.toDouble / 2
      val yRadius = Height.toDouble / 2
      val centerX = XLocation + xRadius
      val centerY = YLocation + yRadius
      if (xRadius == 0.0 || yRadius == 0.0)
         return false
      val normalizedX = centerX - XLocation
      val normalizedY = centerY - YLocation
      return (normalizedX * normalizedX) / (xRadius * xRadius) +
(normalizedY * normalizedY) / (yRadius * yRadius) <= 1.0
   }
}
class Rectangle : Shape() {
   override fun isHit(x: Int, y: Int): Boolean {
      return x >= XLocation && x <= (XLocation + Width) && y >=  YLocation
&& y <= (YLocation  + Height)
   }
}
```

نعدُّ أنَّ الزاوية العلوية اليسرى من لوحة الرسم هي النقطة الصفرية ذات الاحداثيات (0,0)؛ وبالنظر إلى هذين النوعين من الأشكال، سننشئ بعض النسخ منها وسنرى كيف تعمل التعددية الشكلية. سننشئ شكلين بيضويين ومستطيل واحد ثم سنخزِّن هذه النسخ في مجموعة ثم سنعمل على تحديد إن فيما إذا وقعت نقطة ما ضمن أي من هذه الأشكال:

```
fun main(args: Array<String>) {
   val e1 = Ellipsis()
   e1.Height = 10
   e1.Width = 12
```

```
    val e2 = Ellipsis()
    e2.XLocation = 100
    e2.YLocation = 96
    e1.Height = 21
    e1.Width = 19
    val r1 = Rectangle()
    r1.XLocation = 49
    r1.YLocation = 45
    r1.Width = 10
    r1.Height = 10
    val shapes = listOf<Shape>(e1, e2, r1)
    val selected:Shape? = shapes.firstOrNull {shape ->  shape.isHit(50,
52)}
    if(selected == null){
        println("There is no shape at point(50,52)")
    }
    else{
        println("A shape of type ${selected.javaClass.simpleName} has  been
selected.")
    }
}
```

سيطبع تنفيذ الشيفرة البرمجيّة نسخة من شكل المستطيل على الطرفيّة في النقطة المحددة.

ألقِ نظرة باستخدام javap على شيفرة بايتكود الناتجة عن تصريف تلك الشيفرة والتي ستبدو مشابهة للسطرين التاليين (تركت معظمها للتبسيط):

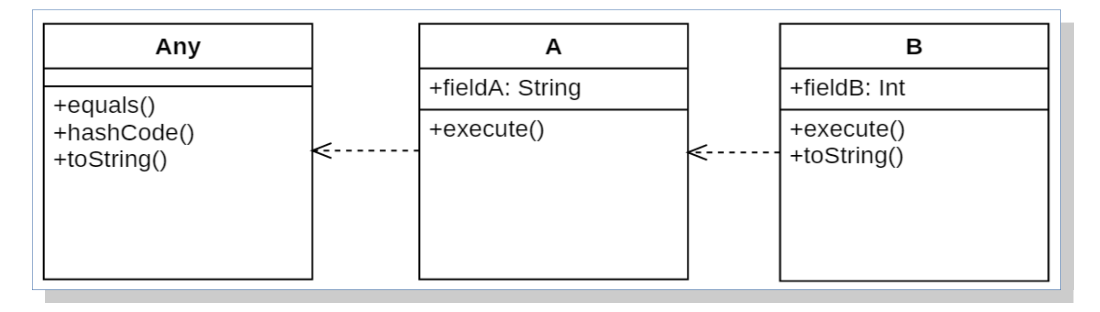
```
169: invokevirtual #69          // Method
com/programming/kotlin/chapter03/Shape.isHit:(II)Z
```

على مستوى بايتكود، هنالك تابع اسمه invokevirtual وظيفته استدعاء دالة افتراضية أُنشِئ بسبب استدعاء شيفرة Rectangle أو Ellipsis، لكن كيف يعرف كيف ومتى يستدعيها؟ ألم استدعي التابع في الصنف Shape؟

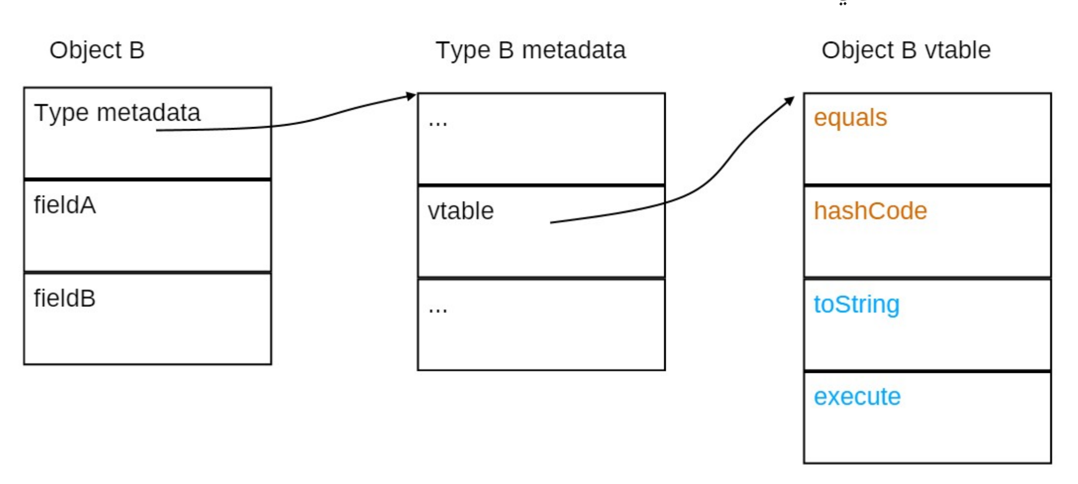
يُعالَج تابع حيوي دِقَّة الوضوح عبر آليّة `vtable` (أي جدول افتراضي [virtual table])، وقد يعتمد النهج الفعلي على تنفيذ JVM، لكنهما سيتشاركان التنفيذ المنطقي نفسه.

عند إنشاء أي نسخة كائن، يُخصَّص له جزء من الذاكرة في منطقة تدعى «الكومة» (heap). الحجم الفعلي من الذاكرة المحجوزة أكبر قليلًا من مجموع الحقول المُخصَّصة، بما في ذلك جميع حقول الأصناف الموروثة وحتى الصنف Any. وسيحصل توقيع وقت التشغيل (runtime footprint) على مساحة إضافيّة أعلى كتلة الذاكرة للاحتفاظ بمرجع إلى معلومات واصف النوع (type descriptor information). ولكل صنف تُعرِّفه، سيكون هنالك كائن محجوز وقت التشغيل. أُضيف هذا المدخل (entry) بوصفه المُدخل الأول لتأمين الموقع دائمًا وضمانه، وبالتالي يجري تجنُّب الحاجة إلى حسابه وقت التشغيل. ويحتوي واصف النوع على قائمة التوابع المُعرّفة مع غيرها من المعلومات المتعلّقة بها، وتبدأ هذه القائمة من الصنف الأعلى في التسلسل الهرمي إلى النوع الفعلي الذي تنتمي النسخ إليه. فالترتيب محدد وهو مثال آخر على التحسين. يُعرَف هذا باسم بنية الجدول الافتراضي ( vtable structure) وهو ليس أكثر من مصفوفة تُشير كل عنصر فيها (الإشارة المرجعيّة) إلى تنفيذ الشيفرة البرمجيّة الحاليّة المحليّة المراد تشغيلها. وأثناء تشغيل البرنامج، سيكون المُصرِّف JIT-er (المصرِّف الآني وقت التنفيذ [just-in time compiler]) المسؤول عن تحويل شيفرة البايتكود التي ينتجها مُصرِّفك إلى شيفرة آلة أو شيفرة تجميع(assembly). إذا قرَّر الصنف المشتق استبدال التابع الافتراضي، سيشير مدخل الجدول الافتراضي (vtable entry) إلى التنفيذ الجديد بدلًا من التنفيذ الذي يوفره آخر صنف في التسلسل الهرمي.

دعنا نتخيَّل أنَّ لدينا صنف `A` يُعرِّف الحقل `fieldA`. هذا الصنف يرث تلقائيًا من الصنف Any. نشتق بعد ذلك هذا الصنف ونضيف حقل إضافي للصنف B الجديد باسم `fieldB`:



يمكنك أن ترى من الرسم البياني السابق أنَّ الصنف A يُعرِّف التابع `execute` والذي يستبدله الصنف المشتق B ويغيِّر تنفيذه؛ وإلى جانب ذلك، يستبدل الصنف B التابع `toString` المُعرَّف في الصنف `Any`. رغم أنَّ هذا مثال بسيط، فإنَّه يحدِّد كيف ستبدو عليه الذاكرة المحجوزة وقت التشغيل. يجب أن تشبه الذاكرة عند إنشاء نسخة من الصنف B المخطط التالي:




متغيِّرك من النوع B ليس إلا مرجعًا إلى كتلة في الذاكرة في منطقة الكومة (heap)، لأنَّ نوع المعلومات (type information) موجود في بداية الكتلة (كما قلنا سابقًا) مع مرجعين إلى قيمة عبر عنوان الذاكرة (indirection، أو مؤشر مرجع إلى قيمة عبر عنوان الذاكرة [pointer dereferencing])، ويمكن أن يُحدِّد موقعها وقت التشغيل بسهولة وسرعة. ويشير الرسم البياني فقط إلى مدخلات `vtable` للنوع `metadata` للتبسيط. ظلَّلت بالألوان التوابع بناءً على على الصنف الذي يوفِّر التنفيذ؛ أول تابعان مُعرَّفان مع تنفيذهما في `Any`، وأمَّا الاثنين المتبقيين فهما معرَّفان مع تنفيذهما في الصنف المشتق B.

إذا نظرت إلى شيفرة البايتكود الناتجة عند استدعاء التابع `execute` عن طريق مرجع للصنف A، ستلاحظ وجود الكلمة المفتاحية الخاصة: `invokevirtual`. وبهذه الطريقة، يستطيع المشغل

الآني تنفيذ الإجراء المعّرف مسبقًا الخاص به لاكتشاف الشيفرة البرمجيّة التي يجب تشغيلها، ولقد وصفنا كل هذا

في وقت سابق.

استنادًا على ما ناقشناه الآن، يمكننا أن نكتشف أنَّ استدعاءً إلى `invokevirtual` يـؤثر سلبًا على المُشغِّل الآني، إذ يجب عليه الحصول على نوع `metadata` أولا ومن هناك، سيُحدِّد vtable ويقفز إلى بداية مجموعة التعليمات التي تمثل شيفرة التجميع للتابع الذي سنستدعيه. وهذا يناقض البرنامج `invokestatic` العادي، إذ لا يلزم تنفيذ مثل هذا التابع المرور على مستويين من مراجع القيمة عبر عنوان الذاكرة (indirection) . إنَّ `Invokestatic` هو برنامج خاص بشيفرة البايتكود وظيفته استدعاء تابع غير افتراضي.

جميع التوابع التي تعرفها الواجهة هي توابع افتراضيّة. وعند استدعاء تابع مثل هذه التوابع لصنف مشتق، فستحصل على معاملة خاصة، فهنالك تابع خاص على مستوى شيفرة البايتكود لمعالجة `this:invokeinterface`. السؤال الذي يطرح نفسه بنفسه هنا، لماذا لا يمكن أن تكون `invokevirtual` بسيطة؟ حسنًا، يحتاج هذا الاستدعاء إلى المزيد من المشاركة بدلًا من اتباع عملية بسيطة من استدعاء تابع افتراضي. يُعدُّ كل مستقبل `invokeinterface` مرجعًا لكائن بسيط، وعلى عكس `invokevirtual`، لا يمكن افتراض موقع `vtable`. وبينما يمكن إنجاز استدعاء إلى `invokevirtual` من خلال مرحلتين أو ثلاث مراحل من مراجع القيمة عبر عنوان الذاكرة (indirection) لاستبيان التابع، ويحتاج استدعاء على مستوى الواجهة أولًا إلى التحقق ما إذا كان الصنف الحالي يُنفِّذ الواجهة وأين، إذا كان الأمر كذلك، موضع هذه التوابع في الصنف الذي يوفر تنفيذًا لها. ولا توجد طريقة بسيطة لضمان ترتيب التوابع في `vtable` لصنفين مختلفيَن ينفذان الواجهة نفسها، ولذلك، في وقت التشغيل، يجب على برنامج شيفرة التجميع المرور على كل عناصر القائمة التي تحوي جميع الواجهات المُنفّذة للبحث عن الهدف؛ وبمجرّد العثور على الواجهة، بسبب `itable` (جدول توابع الواجهة [interface method table])، والذي هو قائمة من التوابع التي تكون بنية إدخالاتها هي نفسها دائمًا لكل صنف يُنفِّذ الواجهة، ويمكن للمشغل الآني (runtime) أن يستمر في استدعاء التابع على أنَّها دالة افتراضيّة؛ ويوجد سبب وجيه لهذا: يمكن أن نملك الصنف A الذي يرث من (يُنفِّذ) الواجهة X والصنف B المشتق من الصنف A؛ يمكن للصنف B أن يستبدل واحد من التوابع المُعرَّفة على مستوى الواجهة.

كما ترى، استدعاءات التابع الافتراضي مكلفة، وهنالك عدد من التحسينات تحتاج آلة جافا الافتراضية JVM إلى تنفيذها لتقصير دورة الاستدعاء، لكن هذه التفاصيل خارج نطاق كتابنا. سأدعك تبحث وحدك إذا كنت فضوليًا،

ومع ذلك، لا تحتاج إلى معرفة هذه المعلومات. القاعدة الأساسيّة هي تجنّب بناء تسلسل هرمي مُعقَّد لصنف مع مستويات عديدة لأنَّ ذلك من شأنه أن يضّر بأداء برنامجك كما قلنا سابقًا.

## 8. قواعد الاستبدال

إذا قرّرت أن يعاد تعريف تابع في صنفك الجديد المشتق من واحد من الأصناف الأصل، فتسمى هذه العملية «بالاستبدال» (overriding)، ولقد استخدمته بالفعل في الفصل السابق. إذا برمجت سابقا بجافا، ستجد أنَّ كوتلن لغة واضحة أكثر؛ ففي جافا، كل تابع هو افتراضي (virtual) افتراضيًا، ولذلك يمكن لأي صنف مشتق استبدال أي تابع موروث؛ أمَّا في كوتلن، فيجب تعريف الدالة على متاحة للاستبدال ليُسمَح بإعادة تعريفها في الأصناف الوارثة؛ ولفعل ذلك، يجب عليك إضافة الكلمة المفتاحية open في بداية تعريف التابع، وعند إعادة تعريف التابع في الصنف الوارث، يجب عليك استعمال الكلمة المفتاحية override معه للإشارة إلى عملية إعادة الاستبدال:

```
abstract class SingleEngineAirplane protected constructor() {
   abstract fun fly()
}

class CesnaAirplane : SingleEngineAirplane() {
  override fun fly() {
     println("Flying a cesna")
  }
}
```

يمكنك دائمًا عدم السماح لأية أصناف وارثة باستبدال الدالة عن طريق إضافة الكلمة المفتاحية final أثناء تعريفها؛ وباستخدام المثال السابق، لا نريد لأيٍّ من وحدات Cesna أن يعيد تعريف التابع fly:

```
class CesnaAirplane : SingleEngineAirplane() {
   final override fun fly() {
      println("Flying a cesna")
   }
```

```
}
```

لا تقتصر على الدوال فقط، فبما أنَّ كوتلن يستعير مفهوم الخاصيات من #C، فيمكنك تحديد الخاصيات على أنَّها افتراضيَّة (virtual):

```
open class Base {
   open val property1: String
      get() = "Base::value"
}
class Derived1 : Base() {
   override val property1: String
     get() = "Derived::value"
}
class Derived2(override val property1: String) : Base() {}
```

يمكنك استبدال خاصيَّة val مع var إذا كان يتطلّب منطق شيفرتك البرمجيّة ذلك، لكن لا يمكن فعل العكس:

```
open class BaseB(open val propertyFoo: String) {
}

class DerivedB : BaseB("") {
   private var _propFoo: String = ""
   override var propertyFoo: String
   get() = _propFoo
   set(value) {
      _propFoo = value
   }
}

fun main(args: Array<String>) {
   val baseB = BaseB("BaseB:value")
   val derivedB= DerivedB()
   derivedB.propertyFoo = "on the spot value"
```

```
    println("BaseB:${baseB.propertyFoo}")
    println("DerivedB:${derivedB.propertyFoo}")
}
```

هنالك سيناريوهات حيث تحتاج إلى الوراثة من صنف وواجهة واحدة على الأقل وكلاهما يعرِّفان وينفِّذان تابع بالاسم نفسه والمعاملات نفسها. في مثل هذه الحالات، تفرض عليك قواعد الوراثة باستبدال التابع. فإذا أنشأت نسخةً جديدةً لكائن واستدعيت التابع المشترك بين الأصناف الموروثة المباشرة، فأي التابعين يجب على المصرّف أن يُنفِّذه؟ ولذلك ستحتاج إلى إزالة الغموض وتوفير التنفيذ، يمكنك استخدام التنفيذ الذي يوفره أحد الصنفين الموروثين أو كلاهما. تخيّل أنك تملك تسلسل هرمي للصنف للتعامل مع مختلف صيغ الصور وتريد توحيدها مع تسلسل هرمي لجهة خارجيّة؛ ولمّا كان كِلا التسلسلات الهرميّة تأتي مع تعريف للدالة save، فستحتاج إلى استبدالها:

```
open class Image {
    open fun save(output: OutputStream) {
        println("Some logic to save an image")
    }
}
interface VendorImage {
    fun save(output: OutputStream) {
        println("Vendor saving an image")
    }
}
class PNGImage : Image(), VendorImage {
    override fun save(output: OutputStream) {
        super<VendorImage>.save(output)
        super<Image>.save(output)
    }
}

fun main(args: Array<String>) {
    val pngImage = PNGImage()
```

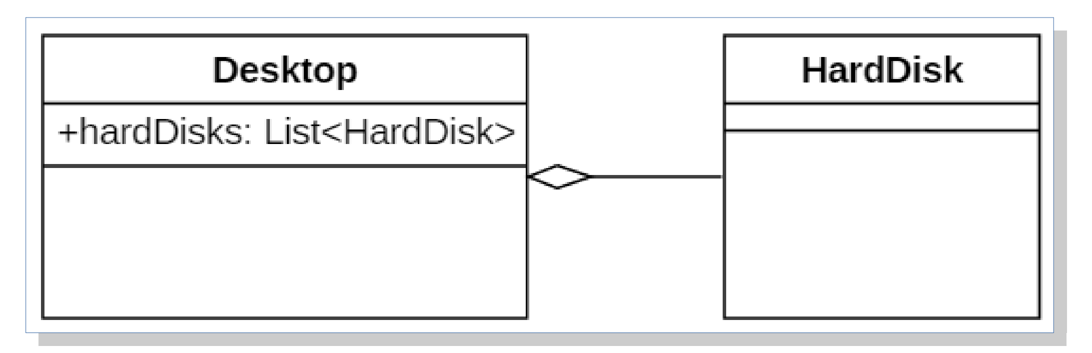
```
    val os = ByteArrayOutputStream()
    pngImage.save(os)
  }
```

لا يُفرَض الاستبدال إذا لم تـوّفر الواجهـة VendorImage التنفيـذ، وتتم الإشـارة إلى تنفيـذ الأصـل عن طريـق super<PARENT>، كما لاحظت في التنفيذ السابق.

## 9. الوراثة مقابل التكوين

واحدة من ميزات اللغة كائنية التوجه هي إعادة استخدام الشيفرة المصدريَّة؛ فبمجـرّد إنشـاء صـنف وتجريبـه، يجب أن يُمثّل كتلة من شيفرة/وظيفة جاهزة للاستخدام.

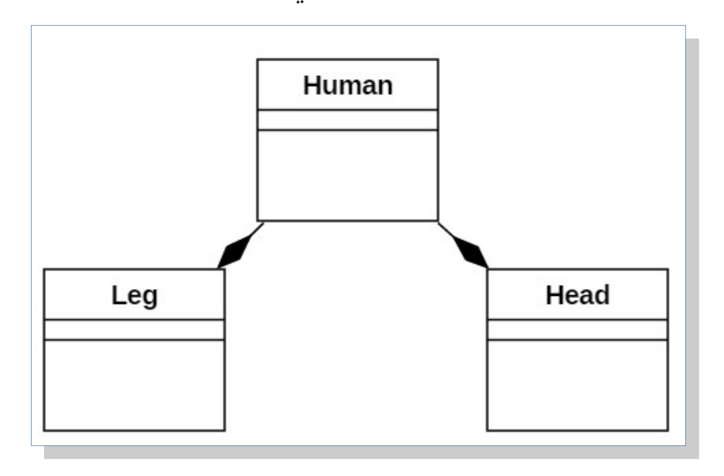
إن أبسط طريقة للاستفادة من صنف مُعرَّف مسبقًا هي إنشاء نسخة منه، لكن يمكنك أيضًا وضع كـائن من ذلـك الصنف داخل صنف آخر جديد. يمكن وضع أي عدد من أنواع الكائنات في الصـنف الجديـد لأداء الوظـائف المطلوبـة. ويسمى مفهوم إنشاء صنف جديد عن طريق استخدام آخر موجـود «بالارتبـاط» (association)، ويشـار إلى هـذا المصطلح بعلاقة «لديه» (has-a). تخيّل أنَّ لديك صنفًا يدعى Desktop ليُمثّل حاسوبًا نموذجيًا لديه قرصًا ثابتًـا، ولوحة أم ...إلخ؛ ولقد استخدمنا هذا المفهوم في أمثلة برمجيّة سابقًا.




يأتي الارتباط بشكلين، وفي العادة يُتغاضـى عن هـذا التفصـيل، فـالنوع الأول من التكـوين (composition) يسمى «التجميع» (aggregation)، وهو يمثل علاقة بين كائنين أو أكثر بشكل يكون لكـل كـائن فيـه دورة حياتـه الخاصة، ولذلك لا يمكن تطـبيق مفهـوم الملكيّـة. وبشـكل أساسـي، يمكن إنشـاء وتـدمير الكائنـات الـتي هي جـزء من

العلاقة بشكل مستقل، كما في المثال السابق للصنف `Desktop`، فيمكن إيقاف الحاسوب دون أي خطأ من القرص الصلب، بينما يمكن التخلص من الحاسوب وأخذ القرص الصلب ووضعه في حاسوب آخر وسيستمر في العمل.

أما النوع الثاني من الارتباط فهو «التكوين» (composition) هو نوع خاص من التجميع؛ ففي هذه الحالة وبمجرد تدمير الكائن الحاوي، ستنتهي الكائنات المحتواة أيضًا. ففي حالة التكوين، ستكون الحاوية مسؤولة عن إنشاء نُسَخ الكائنات المحتواة، ويمكنك التفكير في التركيب بمصطلح مثل «مكون من» أو «جزء من»:



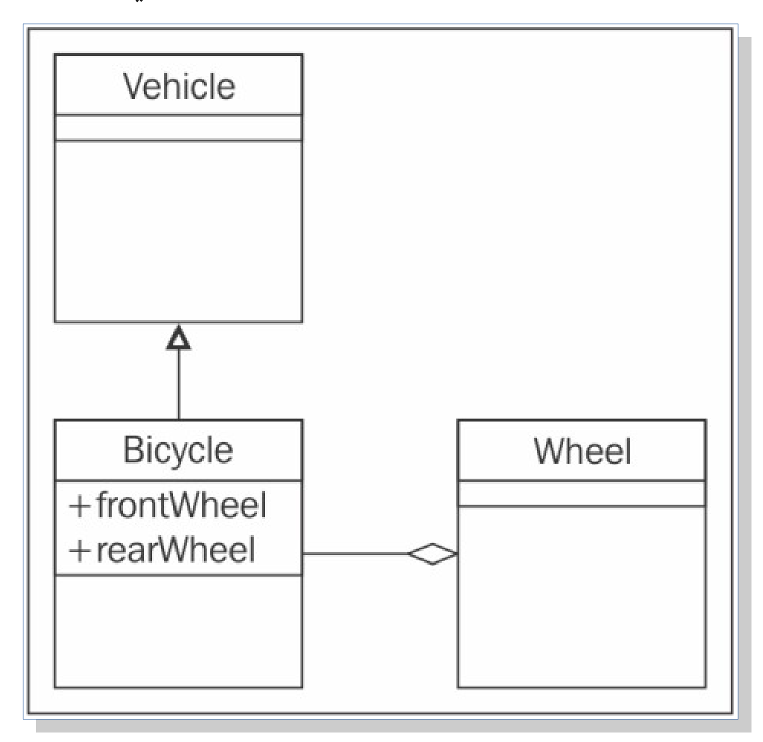

ستتمتع بقدر كبير من المرونة من خلال التكوين. ففي العادة، يَتَطلَّب تصميم التغليف الجيد جعل الكائنات الأعضاء المحتواة في صنفك خاصَّة (private)؛ ولمَّا لن يكن بالإمكان الوصول إلى هذه الكائنات من طرف عميل لصنفك، فلك الحرية في تغييرها عن طريق إضافتها أو حذفها، دون التأثير على الشيفرة البرمجيّة للعميل على الإطلاق. ويمكنك تغيير الأنواع وقت التشغيل (runtime types) لتوفير سلوكيات مختلفة وقت التشغيل إذا كان هنالك حاجة لذلك. فعلى سبيل المثال، نسخة وقت التشغيل للأقراص الصلبة يمكن أن يكون قرص صلب عادي أو من النوع الجديد: قرص تخزين ذو حالة ثابتة (SSD).

يُركَّز في العادة على الوراثة لأهميتها الشديدة في البرمجة كائنيّة التوجه، ويستخدمها مطورو البرامج الجدد في كل مكان، ويمكن أن يؤدي هذا إلى إلى تسلسل هرمي لصنف معقّد وغير ملائم، لذا يجب عليك وضع التكوين في حساباتك عندما ترغب في إنشاء صنف جديد، وفقط إذا كانت قابليّة التطبيق ستجعلك تستعين بالوراثة.

مصطلح آخر يُستخدم بشكل متكرر في عالم OOP وهو is-a وهذا المفهوم مبني بشكل كامل على الميراث،

ولقد رأينا بالفعل الميراث فهو يأتي في شكليّن: الصنف أو الواجهة. وعلاوةً على ذلك، فهو أحادي الاتجاه (على سبيل المثال الدراجة الهوائية هي مركبة لكن المركبة ليست دراجة هوائيّة).

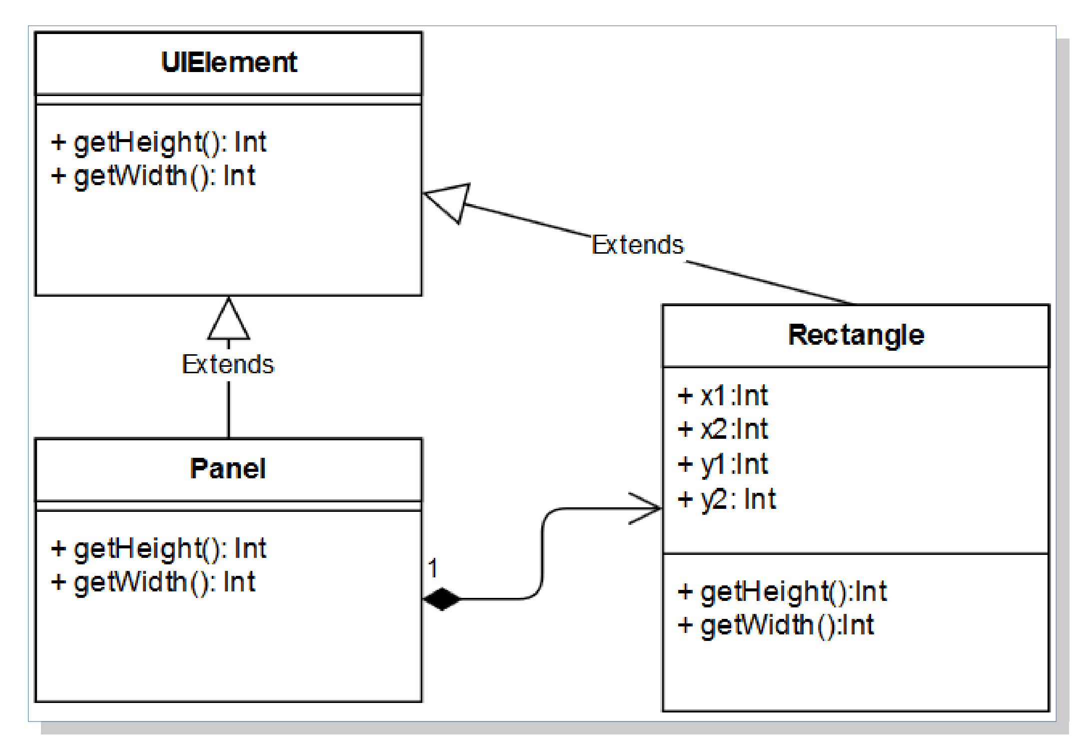
بالطبع، هنالك سيناريوهات أخرى تتطلَّب خلط الارتباط (مهما كان شكله) مع الوراثة. فتخيَّل أنك تبني تسلسلًا هرميًا لصنف يمثّل المركبات، فتبدأ مع الواجهة Vehicle ومن ثم توفّر النوع `Bicycle` الذي سيرث تلك الواجهة وسيضيف، عن طريق التكوين، مرجعين إلى الصنف `Wheel` لتمثيل كما هو موضَّح في الصورة التالية:




## 10. تفويض الصنف

ربما قد سمعت بالفعل عن نمط التفويض (delegation pattern) أو على الأقل استخدمته دون أن تعرف اسمه، فهو يسمح للنوع بتوجيه استدعاء تابع أو أكثر إلى نوع مختلف، ولذلك ستحتاج إلى نوعين لتحقيق ذلك: المُفوِّض والمُفوَّض.

يبدو هـذا مثـل نمـط الوكيل (proxy pattern)، لكنـه ليس كـذلك، فيعـني نمـط الوكيـل توفير بـديل لكـائن موجود لديك، ويتحكم في الوصول إلى الكائن الأصلي لامتلاك السيطرة الكاملة أثناء الوصـول إليـه. فلنفـترض أنـك تكتب إطار واجهة مستخدم (UI framework) وبدأت من صنف مجرَّد يدعى `UIElement`، فكل مكـوّن يُعـرِّف التابعين `getHeight` و `getWidth`.




سترى في الأسفل ترجمة المخطط UML السابق إلى لغة كوتلن، فلقد عرَّفنا الواجهة `UIElement` مع الصـنفين `Panel` و `Rectangle` اللذين يرثانها:

```
interface UIElement {
  fun getHeight(): Int
  fun getWidth(): Int
```

```
}
class Rectangle(val x1: Int, val x2: Int, val y1: Int, val y2: Int) :
UIElement {interface UIElement {
   override fun getHeight() = y2 - y1
   override fun getWidth() = x2 - x1
}
class Panel(val rectangle: Rectangle) : UIElement by rectangle

val panel = Panel(Rectangle(10,100,30,100))
println("Panel height:"+panel.getHeight())
println("Panel witdh:" + panel.getWidth())
```

ربما لاحظت الكلمة المفتاحية `by` في تعريف الصنف `Panel`، فهي ببساطة تلميح للمصرِّف لإنجاز العمل لك: إعادة توجيه الاستدعاءات للتوابع المكشوفة من الواجهة `UIElement` إلى الكائن `Rectangle` الضمني.

من خلال هذا النمط، يمكنك تبديل الوراثة مكان التكوين، ويجب عليك تفضيل التكوين دائمًا

على الوراثة من أجل البساطة والحد من اقتران النوع والمرونة. وباستخدام هذا النهج، يمكن اختيار وتبديل النوع الذي وضعته موضع المُفوِّض بالاستناد إلى المتطلبات المختلفة.

## 11. الأصناف المُغلقَة

الصنف المُغلَق (Sealed classes) في كوتلن هو صنف مُجرَّد، والذي يمكن توسيعه عن طريق الأصناف الفرعيَّة المُعرَّفة على أنَّها أصناف متداخلة داخل الصنف المُغلَق نفسه. بطريقة ما، هذا الخيار أكثر قوّة من خيار التعداد (enumeration)؛ فبطريقة مماثلة للتعداد، تحتوي البنية الهرمية للصنف المُغلَق على مجموعة ثابتة من الخيارات الممكنة؛ ومع ذلك وخلافًا للتعداد، كل خيار مُمثَّل بنسخة واحدة، فيمكن أن تملك الأصناف المشتقة من الصنف المُغلَق على نُسَخ عديدة.

الأصناف المُغلقَة مثاليّة لتعريف أنواع بيانات جبريَّة (algebraic data type)؛ تخيل أنك تريد تصميم هيكل شجرة ثنائيَّة (binary tree structure)، فستفعل شيئًا مشابهًا لما يلي:

```
sealed class IntBinaryTree {
   class EmptyNode : IntBinaryTree()
   class IntBinaryTreeNode(val left: IntBinaryTree, val value: Int, val
right: IntBinaryTree) : IntBinaryTree()
}


...


val tree = IntBinaryTree.IntBinaryTreeNode(
IntBinaryTree.IntBinaryTreeNode(
   IntBinaryTree.EmptyNode(),
   1,
   IntBinaryTree.EmptyNode()),
   10,
   IntBinaryTree.EmptyNode())
```

من الناحيـة المثاليَّة، لا يمكنـك تـثبيت قيمـة الحاويـة (container) في الشـيفرة لتكـون عـددًا صـحيحًا، لكن اجعلها بـدلًا من ذلـك عامَّةً من أجـل قبـول أي نـوع. وبما أننـا لم نتعـرَّف على الأنـواع المُعمَّمة (generics) بعـد، فسنحاول تحري البساطة قدر الإمكان؛ ففي المثال السـابق، ربمـا قـد لاحظت وجـود الكلمـة المفتاحيّـة sealed. إذ محاولة تعريف صنف مشتق من خارج نطاق صنف IntBinaryTree سينتج خطأ أثناء التصريف.

تأتي فائدة استخدام هذه البنية الهرمية للصنف عند استخدامها في تعبير when، فالمصرِّف قادر على اسـتنتاج وتغطيَّة جميع الحالات المحتملة، فعمليـات التحقـق متعبـة. بالنسـبة إلى مطـور سـكالا، سـيبدو هـذا مألوفًا ومشـابهًا لمطابقة النمط (pattern matching). تخيَّل أننا نرغب في كشف عناصر الشجرة إلى قائمـة، وسـتفعل من أجـل ذلك شيئًا مشابهًا للشيفرة التالية:

```
fun toCollection(tree: IntBinaryTree): Collection<Int> = when  (tree) {
   is IntBinaryTree.EmptyNode -> emptyList<Int>()
   is IntBinaryTree.IntBinaryTreeNode -> toCollection(tree.left) +
tree.value + toCollection(tree.right)
}
```

إذا تركت أحد الأصناف المشتقَّة خارج تعبير when، فستحصل على خطأ أثناء تصريف:

```
Error:(12, 5) Kotlin: 'when' expression must be exhaustive, add
necessary 'is EmptyNode' branch or 'else' branch instead.
```

## 12. خلاصة الفصل

بالنسبة لمطور جافا الذي يرغب في الانتقال إلى كوتلن، لقد راجع في هذا الفصل المفاهيم الشائعة، بغض النظر إذا كنت برمجت بلغة كائنية التوجه أم لا. فأنت تعرف الآن المفاهيم الأساسيّة لنهج تصميم البرمجيات ويمكنك كتابة شيفرات برمجية كائنيّة التوجه باستخدام الميزات الجديدة المتاحة في كوتلن وجعلها منظمة وسهلة القراءة. لا أستطيع التأكيد أكثر على أهمية تفضيل التكوين على الوراثة، فلا توجد وصفة قياسية سحرية لإنجاز ذلك بشكل صحيح. يجب أن يكون هدفك دائمًا تبسيط الأشياء ويجب عليك فعل الشيء نفسه عند بناء بنية هرمية للصنف.

سنتعمَّق في الفصل القادم في موضوع الدوال في كوتلن، وسترى كيف أنَّ اللغة قد استعارت من توابع C# الملحقة (C# extension methods) خاصةً التوابع التي تسمح لك بإضافة وظائف جديدة إلى الأصناف الموجودة.

الفصل الرابع:

# الدوال في كوتلن



تعرَّفنا في الفصول السابقة على أساسيات كوتلن وكيفية كتابة شيفرات برمجية إجرائية وكائنيّة التوجه، وسنركِّز في هذا الفصل على الدوال، وما هي الخطوات الأولى في البرمجة الوظيفيّة، وما هي المميزات التي تدعمها كوتلن لجعل برمجة الدوال أسهل.

سنغطي في هذا الفصل المواضيع التالية:

- الدوال والصياغة المختصرة للدوال المجهولة أو الحرفيَّة (function literals).
- الدوال المُوسّعة (Extension functions).
- المعاملات المسماة والمعاملات الافتراضيَّة.
- زيادة تحميل العامل (Operator overloading).
- التعاود (الاستدعاء الذاتي).

## 1. تعريف الدوال

تُعرَّف الدوال باستخدام الكلمة المفتاحية `fun` مع معاملات (parameters) اختياريَّة وقيمة مُعَادة، وإنَّ وجود قائمة المعاملات إجباري، حتى لو لم تُعرِّف أيَّة معاملات؛ فعلى سبيل المثال، الدالة التالية لا تأخذ أيَّة معاملات وتُعيد سلسلة نصيَّة:

```
fun hello() : String = "hello world"
```

كل معامل يكون على شكل `name: type`. تقبل الدالة التالية معاملين من نوع سلسلة نصيّة وتُرجع سلسلة نصيّة أيضًا:

```
fun hello(name: String, location: String): String =
        "hello to you  $name at $location"
```

إذا لم ترجع أي قيمة ذات معنى، ستُعرَّف على أنَّها تُرجع `Unit`.

كما تحدثنا في الفصل الثاني، أساسيات كوتلن، يُشبه `Unit` النوع `void` في سي وجافا، وبالتالي سيكون نظام

الأنواع في كوتلن منتظم باستخدام صنف يُعدُّ جزءًا من البنية الهرمية للأنواع (بدلًا من نوع خاص مثل `void`)، فكل دالة يجب أن تُرجع قيمة، وهذه القيمة قد تكون `Unit` أو لا.

يمكن للمطوّر حذف نوع الإرجاع في الدوال التي تُرجِع القيمة `Unit` لصياغة نمط الإجراء وإليك مثالٌ عن ذلك:

```
// الدالتان التاليتان متماثلتان تمامًا ولا يوجد أي فرق بينهما
fun print1(str: String): Unit {
   println(str)
}

fun print2(str: String) {
   println(str)
}
```

## 2. الدوال وحيدة التعبير

يجب أن تصرِّح الدالة عادةً عن نوع البيانات التي تعيدها، لكن هنالك استثناء للدوال التي تتكوّن من تعبير واحد، وغالبًا ما يُشار إليها بدوال السطر الواحد (one line functions) أو الدوال وحيدة التعبير (Single expression functions)، ويمكن لهذه الدوال استخدام صياغة مختصرة بحذف الأقواس المعقوصة `{}` واستخدام رمز المساواة = قبل التعبير بدلًا من الكلمة المفتاحية `return`:

```
fun square(k: Int) = k * k
```

لاحظ كيف أنَّ الدالة لا تحتاج إلى التصريح عن النوع المعاد الذي هو `int` مسندةً عملية استنتاج النوع للمُصرِّف. والأساس المنطقي وراء هذه الميّزة هي أنَّ الدوال القصيرة سهلة القراءة، واستنتاج القيمة المعادة بسيطٌ ولا يتطلب كتابة شيفرة طويلة؛ ومع ذلك، يمكنك دائمًا التصريح عن نوع القيم المعادة إذا كنت تعتقد أنَّ الشيفرة ستكون أوضح:

```
fun square2(k: Int): Int = k * k
```

يمكنـك دائمًا إذا رغبت كتابـة الـدوال وحيـدة التعبـير بالنمـط العـادي؛ فعلى سـبيل المثـال، الـدالتان التاليتـان متطابقتان تمامًا وتُصرَّفان إلى شيفرة بايتكود نفسها:

```
fun concat1(a: String, b: String) = a + b
fun concat2(a: String, b: String): String {
   return a + b
}
```

**ملاحظة**

يفرض المصرّف قاعدةً وهي أنَّ الدالة وحيدة التعبير هي من تستطيع حذف نوع القيمة المعادة فقط.

## 3. الدوال التابعة للأصناف

يسمى النوع الأول من الدوال بالدوال التابعة (member functions)، وتُعرَّف هذه الدوال داخل صـنف أو كـائن أو واجهـة، وتُسـتدعَى هـذه الدالـة باسـتخدام اسـم الصـنف أو الكـائن الحـاوي مـع نقطـة متبوعـةً باسـم الدالـة والمعاملات بين قوسين؛ فعلى سـبيل المثـال، إن أردت اسـتدعاء دالـة اسـمها take مـع نسـخة من النـوع String، فيمكنك صياغة هذا الاستدعاء بالشكل التالي:

```
val string = "hello"
val length = string.take(5)
```

يمكن أن تُشير الدوال التابعة إلى أنفسها دون الحاجة إلى اسم النسـخة، وهـذا يعـود إلى أنَّ اسـتدعاءات الدالـة تعمل على النسخة الحالية، ويُشار إليها على النحو التالي:

```
object Rectangle {
   fun printArea(width: Int, height: Int): Unit {
      val area = calculateArea(width, height)
      println("The area is $area")
   }
```

```
    fun calculateArea(width: Int, height: Int): Int {
      return width * height
    }
}
```

يعرض مقتطف الشيفرة المصدريّة دالتيّن تحسب إحداهما، أي الدالة `calculateArea`، مساحة المستطيل وتطبعه على الطرفيّة، وتأخذ الدالة الأخرى، أي `printArea`، معامليّن هما: `width` الذي يمثِّل العرض و `height` الذي يمثِّل الطول ثم تستدعي الدالة `calculateArea` (لاحظ استعمال اسم الدالة دون الإشارة إلى اسم النسخة الحاوية) لحساب المساحة وطباعتها.

ستلاحظ أيضًا أن دالة `calculateArea` تستخدم `return`، لأنَّ قيمتها ستُستخدَم في دوال أخرى وأما الدالة `printArea` فلا تملك أي قيمة مجديّة تعيدها، ولذلك صرَّحنا على أنَّ القيمة المعادة هي `Unit`.

## 4. الدوال المحليَّة

فكرة الدوال بسيطة للغاية: تقسيم برنامجك إلى أجزاء أصغر يمكن تفسيرها بسهولة والسماح بإعادة استخدامها لأداء الوظيفة نفسها في برنامج تجنبًا للتكرار، وهذه النقطة الثانيّة تُعرف بمبدأ: «لا تكرر نفسك» (DRY اختصارٌ للعبارة Don't Repeat Yourself)؛ فكلما زادت عدد المرات التي تكتب فيها نفس الشيفرة البرمجية، ازدادت نسبة الأخطاء.

الملخص المنطقي لهذا المبدأ هو إنشاء برنامج يتكون من عدة دوال صغيرة تنجز كل واحدة منها مهمة ما. وهذا شبيه لمبدأ يونكس للبرامج الصغيرة، إذ يُنفِّذ كل برنامج وظيفة واحدة.

يُطبَّق المبدأ نفسه على الشيفرة البرمجيّة داخل الدالة؛ فيمكن عادةً في لغة جافا تقسيم دالة أو تابع كبير إلى أجزاء عن طريق استدعاء عدة دوال داعمة مُعرَّفة إما في الصنف نفسه أو في صنف مساعد (helper class) يحتوي على توابع ثابتة.

تسمح لنا كوتلن باتخاذ خطوة إضافيَّة من خلال دعم الدوال المُعرَّفة داخل دوال أخرى، وتسمى هذه الدوال بالدوال المحلية (local functions) أو الدوال المتشعبة (nested functions) أو متداخلة، ويمكن أن تكون

الدوال متداخلة على عدة مستويات.

يمكن كتابة المثال السابق لطباعة المساحة بالنمط التالي:

```
fun printArea(width: Int, height: Int): Unit {
   fun calculateArea(width: Int, height: Int): Int = width * height
   val area = calculateArea(width, height)
   println("The area is $area")
}
```

كما ترى، فإنَّ الدالة `calculateArea` أصبحت داخل `printArea` وبالتالي لا يمكن الوصول إليها من خارج شيفرتها البرمجيَّة، ويفيدنا هذا عندما نريد إخفاء الدوال التي تُستخدَم لإنجاز أمرٍ ما داخل دالة أكبر، ويمكننا تحقيق تأثير مشابه بتعريف دالة تابعة على أنَّها خاصة (private).

هل تملك الدوال المحليَّة ميّزات أخرى؟ نعم، يمكن للدوال المحليَّة الوصول إلى المعاملات والمتغيرات المُعرَّفة خارج نطاقها:

```
fun printArea2(width: Int, height: Int): Unit {
   fun calculateArea(): Int = width * height
   val area = calculateArea()
   println("The area is $area")
}
```

لاحظ أننا حذفنا المعاملات من الدالة `calculateArea` لأنه أصبح بإمكانها الآن استخدام معاملات الدالة `printArea2` الموجودة ضمن نطاقها مباشرةً، ويُسهِّل هذا من قراءة الدالة المتداخلة ويوفر عناء تكرار تعريف المعاملات، وهو أمر مفيد للغاية خصوصًا مع الدوال التي تملك معاملات عديدة.

إليك مثال لدالة يمكن تقسيمها إلى دوال محليَّة:

```
fun fizzbuzz(start: Int, end: Int): Unit {
   for (k in start..end) {
      if (k % 3 == 0 && k % 5 == 0)
         println("Fizz Buzz")
```

```
      else if (k % 3 == 0)
        println("Fizz")
      else if (k % 5 == 0)
        println("Buzz")
      else
        println(k)
    }
  }
```

هذه هي مشكلة Fizz Buzz الشهيرة: المطلوب منك طباعة الأعداد الصحيح من قيمة start إلى قيمة end، وستطبع الكلمة Fizz إذا كان العدد من مضاعفات 3 أو تطبع الكلمة Buzz إذا كان من مضاعفات 5، أو تُطبَع الكلمتان Fizz Buzz معًا إذا كان من مضاعفات 3 و 5 في الوقت نفسه.

الحل الأول قصير وسهل القراءة لكنه يُكرِّر بعض الشيفرات البرمجيّة، إذ يُتحقَّق من باقي القسمة مرَّتين مما يضاعف من فرصة وجود مشكلةٍ.. ومن الواضح أنَّ هذا المثال بسيط، لذلك فرصة حدوث خطأ في الصياغة ضئيلة، لكنه يعمل على توضيح مشكلة للبرامج الأكبر.

يمكننا التصريح عن دالة محليّة تُنجِز عملية التحقق من باقي القسمة، لذلك لا يتعيَّن علينا سوى برمجة هذه الوظيفة مرةً واحدةً، ويقودنا هذا إلى الإصدار الثاني من برنامجنا السابق:

```
fun fizzbuzz2(start: Int, end: Int): Unit {
  fun isFizz(k: Int): Boolean = k % 3 == 0
  fun isBuzz(k: Int): Boolean = k % 5 == 0

  for (k in start..end) {
    if (isFizz(k) && isBuzz(k))
      println("Fizz Buzz")
    else if (isFizz(k))
      println("Fizz")
    else if (isBuzz(k))
      println("Buzz")
```

```
      else
         println(k)
   }
}
```

هنا، تستدعي أفرع `if...else` الدالتين `isFizz` و `isBuzz` المحليَّتين.

ومع ذلك، إن تمرير k للدالة كل مرة هو شيء طويل قليلًا، فهل هنالك طريقة لتجنب هذا؟ نعم، يمكننا تعريف الدوال المحليَّة ليس فقط من داخل الدوال الأخرى، لكن أيضًا في حلقات `for` و `while` وبقيَّة الكتل:

```
fun fizzbuzz3(start: Int, end: Int): Unit {
   for (k in start..end) {

      fun isFizz(): Boolean = k % 3 == 0
      fun isBuzz(): Boolean = k % 5 == 0

      if (isFizz() && isBuzz())
         println("Fizz Buzz")
      else if (isFizz())
         println("Fizz")
      else if (isBuzz())
         println("Buzz")
      else
         println(k)
   }
}
```

هذه المرَّة، نقلنا تعريف الدالتين إلى داخل حلقة `for`، واستطعنا حتى الآن حذف إعلان المعامل والوصول إلى المتغير kمباشرةً.

وأخيرًا، يمكننا الاستفادة من عبارة when التي تحدثنا عنها في الفصل الثاني، أساسيات كوتلن، لحذف بعض كلمات `if..else`:

```
fun fizzbuzz4(start: Int, end: Int): Unit {
   for (k in start..end) {

      fun isFizz(): Boolean = k % 3 == 0
      fun isBuzz(): Boolean = k % 5 == 0
      when {
         isFizz() && isBuzz() -> println("Fizz Buzz")
         isFizz() -> println("Fizz")
         isBuzz() -> println("Buzz")
         else -> println(k)
      }
   }
}
```

يعطينا هذا الحل النهائي الذي يتجنّب تكرار التعليمات البرمجية ويُسهِّل من قراءة الشيفرة وصيانتها وتعديلها لاحقًا.

## 5. دوال المستوى الأعلى

تدعم كوتلن - بالإضافة إلى الدوال التابعة والدوال المحليِّة - إمكانية التصريح عن دوال في أعلى مستوى (top-level functions)، وتكون هذه الدوال موجودةً خارج أي صنف أو كائن أو واجهة وتُعرَّف مباشرةً داخل الشيفرة المصدرية، ويأتي اسم «أعلى مستوى» (top-level) من حقيقة أنَّ الدوال غير متداخلة داخل أي بُنيَة (structure) ولذلك فهي في أعلى البنية الهرمية للأصناف والدوال.

تفيد الدوال الموجودة في أعلى مستوى في تعريف دوال مساعدة (helper functions) أو دوال خدمية (utility functions)، فليس تجميعها مع دوال أخرى منطقيًا بالضرورة عندما لا يضيف الكائن الحاوي أي قيمة. في جافا، تتصف هذه الأنواع من الدوال على أنها دوالًا ساكنةً (static function) داخل الأصناف المساعدة (helper classes). وخير مثال على ذلك دوال الحزمة `collections` في مكتبة جافا القياسيَّة.

ومع ذلك، بعضُ الدوال مستقلةٌ ومن غير المنطقي إنشاء كائن ليحتويها وأفضل مثال على هذه الدوال هو

الدالة require من مكتبة كوتلن القياسيَّة والتي تُستخدَم للتأكد من أنَّ المعاملات تلبي شروطًا غير ثابتة عند استعمالها؛ فعلى سبيل المثال، إذا كان يجب أن يكون المعامل أكبر من 10 دائمًا، فيمكننا الاستفادة من الدالة require بالشكل التالي :

```
fun foo(k: Int) {
   require(k > 10, { "k should be greater than 10" })
}
```

يمكن وضع هذه الدالة وأخواتها، check و error و requireNotNull، داخل كائن يسمى Assertions (التوكيد، أو أي اسم آخر بالمعنى نفسه)، لكن لا تضيف ذلك أي قيمة. ويمكننا ترك مثل هذه الدوال في أعلى مستوى وذلك بتعريفها مباشرةً داخل ملف يدعى assertions.kt.

## 6. المُعامِلات المسماة

المعاملات المسماة (Named parameters) هي عبارة عن إعطاء اسم لكل معامل من معاملات الدالة أثناء تمريرها إليها، وفائدة هذه الميزة هي وضوح وظيفة كل معامل من اسمه وتسهيل قراءة الشيفرة أيضًا خصوصًا في الدوال التي تملك العديد من المعاملات.

في المثال التالي، نتحقق مما إذا كانت السلسلة الأولى تحتوي على سلسلة فرعيَّة من السلسلة الثانيَّة:

```
val string = "a kindness of ravens"
string.regionMatches(14, "Red Ravens", 4, 6, true)
```

لاستخدام المعاملات المسماة، نضع اسم المعامل قبل القيمة، وهذا استدعاء للوظيفة مرةً أخرى مع المعاملات المسماة:

```
string.regionMatches(thisOffset = 14, other = "Red Ravens",  otherOffset =
4, length = 6, ignoreCase = true)
```

المثال الثاني أسهل قراءةً وأكثر دقّةً، إذ أصبحت المعاملات الآن واضحة، ويمكنك الآن تخمين فائدة المتغير المنطقي الأخير الذي هو حساسيّة حالة الأحرف؛ وإذا لم تكن المعاملات مسماة في الدالة، فيجب عليك التحقق من توثيقها أو من شيفرة تعريفها لمعرفة وظيفة كل معامل مُمرَّر.

فائدة أخرى تبزر للمعاملات المسماة في الدوال التي تملك عدة معاملات من النوع نفسه هي انخفاض نسبة الخطأ عند ربط القيمة بالاسم. ففي المثال القادم، سترى كيف أنَّ الدالة التي تقبل معاملات منطقيّة متعددة يمكن خطأ تبديل معاملاتها عند عدم تسميتها:

```
fun deleteFiles(filePattern: String, recursive: Boolean,  ignoreCase:
Boolean, deleteDirectories: Boolean): Unit
```

وازن بين الطريقتيّن المختلفتيّن لاستدعاء هذه الدالة:

```
deleteFiles("*.jpg", true, true, false)
deleteFiles("*.jpg", recursive = true, ignoreCase = true,
deleteDirectories = false)
```

هل لاحظت عدم تسميّة المعامل الأول ووضعنا أسماء للآخرين؟ عند استدعاء دالة، لا تحتاج إلى تسميّة كل المعاملات، فالقاعدة بسيطة: «عند تسميّة معامل، يجب تسميّة جميع المعاملات التي تليه أيضًا».

تسمح لك المعاملات المسماة بتغيير ترتيب المعاملات أيضًا لتناسب الاستدعاء أيضًا؛ فعلى سبيل المثال، طريقتا استدعاء الدالة التابعة endsWith متطابقين في المثال التالي:

```
val string = "a kindness of ravens"
string.endsWith(suffix = "ravens", ignoreCase = true)
string.endsWith(ignoreCase = true, suffix = "ravens")
```

سنوضّح لماذا هذا مفيد في القسم التالي في المعاملات الافتراضيَّة. فتغيير ترتيب المعاملات يسمح لنا باختيار أي من المعاملات الافتراضيّة نريد استبدالها.

**ملاحظة**: يمكن استخدام المعاملات المسماة في دوال كوتلن المعرّفة وليس في دوال جافا المعرّفة.

وهذا بسبب أن شيفرة جافا لا تحتفظ دائمًا بأسماء المعاملات عند تصريفها إلى بايتكود.

## 7. المُعامِلات الافتراضيَّة

في بعض الأحيان، من الأفضل توفير قيم افتراضيَّة للمعاملات في دالة ما لاستخدامها عند عدم تمرير قيمة لها. دعنا نقول أنَّنا نريد إنشاء مجمَّع خيوط (thread pool)، ويمكن أن تكون القيمة الافتراضيَّة لعدد الخيوط هو عدد أنوية وحدة المعالجة المركزيّة CPU، ويمكن للمستخدم تغيير ذلك إذا أراد.

وطريقة تحقيق ذلك في اللغات التي لا تدعم المعاملات الافتراضيَّة هي تقديم إصدارات معاد تعريفها من الدالة نفسها[6]:

```
fun createThreadPool(): ExecutorService {
   val threadCount = Runtime.getRuntime().availableProcessors()
   return createThreadPool(threadCount)
}

fun createThreadPool(threadCount: Int): ExecutorService {
   return Executors.newFixedThreadPool(threadCount)
}
```

يمكن للمستخدم هنا اختيار استدعاء أي إصدار، ومع ذلك، يعني عدد المعاملات أحيانًا أنَّنا نملك عدة إصدارات معاد تعريفها من الدالة نفسها، مما يؤدي إلى تكرار الشيفرة بدون حاجة. فعلى سبيل المثال، تملك المكتبة القياسيَّة `BigDecimal` الإصدارات التالية لدالة واحدة فقط:

```
public BigDecimal divide(BigDecimal divisor)
public BigDecimal divide(BigDecimal divisor, RoundingMode  roundingMode)
public BigDecimal divide(BigDecimal divisor, int scale,  RoundingMode
roundingMode)
```

6 تدعى هذه الإصدارات ببصمات الدالة (function signature) وتختلف كل بصمة عن أخرى بعدد المعاملات أو نوعها (أو كلاهما).

وهنالك العديد من الإصدارات الأخرى، فكل إصدار يفوِّض القيم إلى الإصدار الآخر مع قيمة افتراضيّة حسَّاسة.

في كوتلن، يمكن تحديد قيم افتراضيّة لمعاملات دالة واحدة أو أكثر، والتي يمكن استخدامها عند عدم تحديد المعاملات، ويسمح لنا هذا بتعريف دالة واحدة لحالات متعدِّدة، وبالتالي تجنّب الحاجة إلى إصدارات متعدِّدة زائدة للدالة.

سنكتب مثلًا الدالة `divide` مرة أخرى ونجمع جميع إصداراتها في إصدار واحد عبر استخدام المعاملات الافتراضيّة:

```
fun divide(divisor: BigDecimal, scale: Int = 0, roundingMode:
RoundingMode = RoundingMode.UNNECESSARY): BigDecimal
```

عند استدعاء هذه الدالة، يمكننا تجاهل بعض أو جميع المعاملات، ولكن بمجرَّد حذف معاملٍ، يجب حذف جميع المعاملات التي تليها. فعلى سبيل المثال، يمكننا استدعاء هذه الدالة بهذه الطرق:

```
divide(BigDecimal(12.34))
divide(BigDecimal(12.34), 8)
divide(BigDecimal(12.34), 8, RoundingMode.HALF_DOWN)
```

لكننا لا يمكننا استدعاءها بالشكل التالي:

```
divide(BigDecimal(12.34), RoundingMode.HALF_DOWN)
```

ومع ذلك، لحل هذه المشكلة، يمكننا خلط المعاملات المسماة والمعاملات الافتراضيّة:

```
divide(BigDecimal(12.34), roundingMode = RoundingMode.HALF_DOWN)
```

بشكل عام، إنَّ استخدام المعاملات المسماة بالاشتراك مع المعاملات الافتراضيّة هو شيء مفيد وفعَّال للغاية، إذ يُتيح لنا توفير دالة واحدة، ويمكن للمستخدمين استبدال القيم الافتراضيّة التي يرغبون بها.

| ملاحظة | عند استبدال دالة تحتوي على معاملات افتراضيّة، يجب علينا الحفاظ على نفس إصدار (بصمة) الدالة. |
|---|---|

يمكن استـخدام المعـاملات الافتراضـيَّة في البانيـات (constructors) لتجنّب الحاجـة إلى بانيـات ثانويَّة متعدِّدة. ويوضّح لك المثال التالي بانيات متعدِّدة:

```
class Student(val name: String, val registered: Boolean, credits:  Int) {
   constructor(name: String) : this(name, false, 0)
   constructor(name: String, registered: Boolean) : this(name,
registered, 0)
}
```

يمكن إعادة صياغة هذه البانيات كما يلي:

```
class Student2(val name: String, val registered: Boolean = false,
credits: Int = 0)
```

## 8. الدوال الملحقة المُوسِّعة

تصادفك في الكثير من الأحيان حالات يمكن أن يستفيد نوعٌ لا تملك سيطرةً عليه من دالة إضافيَّة، فربمـا كنت تتمنى دائمًا أن يملك النوع String الدالة reverse() أو ربما كنت تقول في نفسك لمَ لا يملك النـوع list دالـةً مثل drop التي تُرجع لك نسخةً من القائمة مع حذف أول س عنصر منها.

سيكون النهج كائني التوجيه نهجًا يُستعمَل لتوسيع عمل الأنواع عبر إنشاء نوع فرعي يضيف الدوال المطلوبة:

```
abstract class DroppableList<E> : ArrayList<E>() {
   fun drop(k: Int): List<E> {
      val resultSize = size - k
      when {
         resultSize <= 0 -> return emptyList<E>()
         else -> {
            val list = ArrayList<E>(resultSize)
            for (index in k..size - 1) {
               list.add(this[index])
            }
            return list
```

```
            }
        }
    }
}
```

لكن هذا ليس ممكنًا دائمًا، فالصنف الذي أضيفت الكلمة المفتاحية `final` (ثابت أو نهائي) إلى تعريفه لا يمكن توسيعه، وقد لا تتحكم بالنسخة عند اشتقاقها منه، لذلك لا يمكنك وضع النوع الفرعي مكان النوع الموجود.

الحل النموذجي هو إنشاء دالةٍ في صنف منفصل تقبل تمرير النسخة ضمن معاملاتها. على سبيل المثال في جافا، من الشائع جدًا رؤية أصنافٍ تتكوَّن بالكامل من دوال مساعدة لاستعمالها مع شتَّى النُسَخ مثل الصنف `java.util.Collections` الذي يحتوي على العشرات من الدوال الثابتة التي تقدّم وظائف متعددة تساعد في التعامل مع التجميعات (collections):

```
fun <E> drop(k: Int, list: List<E>): List<E> {
    val resultSize = list.size - k
    when {
        resultSize <= 0 -> return emptyList<E>()
        else -> {
            val newList = ArrayList<E>(resultSize)
            for (index in k..list.size - 1) {
                newList.add(list[index])
            }
            return newList
        }
    }
}
```

مشكلة هذا الحل تتفرع إلى فرعين: أولًا، لا يمكننا استخدام الإكمال التلقائي للشيفرة التي توفرها بيئة التطوير المتكاملة (IDE) لنعرف الدوال المتاحة، وهذا لأنَّنا نكتب اسم الدالة في البداية؛ ثانيًا، إذا كان لدينا عدَّة دوال نرغب في استدعائها سويةً، فسينتهي بنا الأمر بشيفرة برمجيَّة صعبة القراءة مثل هذه:

```
reverse(take(3, drop(2, list)))
```

ألن يكون من الأفضل لو تمكنًا من الوصول إلى النسخة list لاستدعاء التوابع معها مباشرةً مثل الاستدعاء التالي؟!

```
list.drop(2).take(3).reverse()
```

تسمح لنا الدوال المُوسّعة (extension functions) بتحقيق ذلك دون الحاجة إلى إنشاء نوع فرعي أو تعديل النوع الأصلي أو الالتفاف حول الصنف.

يُعلَن عن دالة موسّعة بتعريف دالة في المستوى الأعلى مثل المعتاد، لكن مع إلحاق اسم النوع المطلوب باسم الدالة؛ يسمى نوع النسخة الذي سيُستخدَم مع تلك الدالة «بالنوع المستقبل» (receiver type)، ويقال أن النوع المُستقبِل مُوسَّع مع الدالة المُوسّعَة تلك.

لنعود للدالة drop مرَّة أخرى، ولكن سنحاول هنا تطبيقها على أنَّها دالة مُوسّعَة:

```
fun <E> List<E>.drop(k: Int): List<E> {
   val resultSize = size - k
   when {
      resultSize <= 0 -> return emptyList<E>()
      else -> {
         val list = ArrayList<E>(resultSize)
         for (index in k..size - 1) {
            list.add(this[index])
         }
         return list
      }
   }
}
```

لاحظ استخدام الكلمة المفتاحية this داخل جسم الدالة التي تُستخدَم للإشارة إلى النسخة المستقبلة، أي الكائن الذي استدعيَت الدالة معه؛ وعندما نكون داخل دالة المُوسّعة، تُشير this دائمًا إلى النسخة المستقبلة،

وتحتاج النُسَخ خارج هذا النطاق إلى التأهيل.

لاستخدام دالة مُوسّعة، نستوردها كما نفعل مع أي دالة من المستوى أعلى باستخدام اسم الدالة والحزمة الـتي تتواجد فيها:

```
import com.packt.chapter4.drop
val list = listOf(1,2,3)
val droppedList = list.drop2(2)
```

### أ. 1.8.4 أولوية الدالة المُوسّعة

لا يمكن لدوال مُوسّعة استبدال الدوال المُعرَّفة في صنفٍ أو واجهةٍ. إذا تماثلت بصمة دالـة موسّـعة مـع بصـمة إحـدى دوال صـنف أو واجهـة مـا (تطـابقت في الاسـم وأنـواع المعـاملات وترتيبهـا، والنـوع المعـاد)، فلن يسـتدعيها المُصـــــرِّف أبـــــدًا بـــــل سيســـــتدعي دالـــــة الصـــــنف أو الواجهـــــة تلـــــك الـــــتي لها الأولوية.

عندما يَعثُر المُصرِّف -أثناء عملية التصريف- على استدعاء دالة، يبحث أولًا في الدوال التابعة المُعرَّفة في نـوع النسخة ثم في الدوال التابعة المُعرَّف في الأصناف العليا والواجهات، وإذا عَثَر على الدالة التابعة المنشودة، فسيربط عملية الاستدعاء بها.

وإن لم يعـثر المُصـرِّف على أي دالـة تابعـة مطابقـة، يبحث بعدئـذٍ في أي اسـتيراد مُوسّـع للشـيفرة والواقـع في نطاقها. افترض وجود التعريفات التاليّة:

```
class Submarine {
   fun fire(): Unit {
      println("Firing torpedoes")
   }

   fun submerge(): Unit {
      println("Submerging")
   }
}
```

```
fun Submarine.fire(): Unit {
   println("Fire on board!")
}

fun Submarine.submerge(depth: Int): Unit {
   println("Submerging to a depth of $depth fathoms")
}
```

نملك هنا النوع Submarine مع الدالتين fire() و submerge() ولقد عرَّفنا أيضًا دالتين مُوسِّعتَين Submarine باسم يطابق الدالتين التابعتين الأساسيتين. سنستخدم التعليمات التالية إذا أردنا استدعاء هذه الدوال:

```
val sub = Submarine()
sub.fire()
sub.submerge()
```

سيكون الناتج FiringTorpedoes و Submerging، إذ نلاحظ أنَّ المُصرِّف ربط الاستدعاء بالدالة fire() المُعرَّفة في الصنف submarine. لا يمكن في هذا المثال استدعاء إحدى الدالتين الموسعتين أبدًا لعدم وجود وسيلة لفكها من الدالة التابعة للصنف.

ومع ذلك، فإنَّ الدالة submerge() لها بصمة مختلفة، لذلك يمكن للمُصرِّف ربطها بالاعتماد على عدد المعاملات المستخدمة:

```
val sub = Submarine()
sub.submerge()
sub.submerge(10)
```

المخرجات هي:

```
Submerging
Submerging to a depth of 10 fathoms.
```

### ب. الدوال المُوسّعة مع null

تدعم كوتلن توسيع الدوال مع قيم العَدم null. تشير الإشارة المرجعية this في هذه الحالات إلى قيمة العَدم null، ولذلك فإن الدالة Any التي لا تُعالج بأمان المرجع الذي يشير إلى القيمة null سترمي استثناء مؤشر العَدم (null pointer exception).

تُمثِّل هذه الوظيفةُ كيفيَّة إعادة تعريف دوال التحقق من المساواة لتوفير الاستخدام الآمن لها حتى لقيم العَدم:

```
fun Any?.safeEquals(other: Any?): Boolean {
   if (this == null && other == null) return true
   if (this == null) return false
   return this.equals(other)
}
```

### ت. 3.8.4 الدوال المُوسّعة التابعة

يُصرَّح عن الدوال المُوسّعة في المستوى الأعلى، لكن يمكننا تعريفها توابع داخل الأصناف، ويمكن استخدام هذه الطريقة إذا أردنا الحد من نطاقها:

```
class Mappings {
   private val map = hashMapOf<Int, String>()
   private fun String.stringAdd(): Unit {
      map.put(hashCode(), this)
   }

   fun add(str: String): Unit = str.stringAdd()
}
```

في هذا المثال، عرَّفنا دالةً مُوسّعة تُضيف سلسلة نصيَّة إلى hashmap، إذ تستدعي الدالة الثانية الدالة المُوسّعة فقط؛ وتُشير طريقة إضافة hashmap إلى كيفيَّة عمل المستقبلات في الدالة المُوسّعة التابعة.

تُعرَّف الدالة hashCode في Any، وهي موروثةٌ أيضًا في الصنف Mappings و String. عند استدعاء

hashCode في دالـة مُوسّـعة، يوجـد دالتـان محتملتـان واقعتـان في نطـاق اسـتخدامها؛ تسـمى الدالـة الأولى في النسـخة Mappings بالمسـتقبل الموفـد (dispatch receiver)، وتسـمى الدالـة الثانيـة في النسـخة String بالمستقبل المُوسّع (extension receiver).

عندما وجود هذا النوع من تظليل الاسم، فـإن المُصرِّف يسـتعمل بشـكل افتراضـي المسـتقبل المُوسَّـع، لـذا في المثال السابق، شيفرة hashCode التي ستُستخدَم هي لنسخة String، ولاستخدام المستقبل الموفد، يجب علينـا استخدام المؤهل التالي:

```
class Mappings {
  private val map = hashMapOf<Int, String>()

  private fun String.stringAdd(): Unit {
    map.put(this@Mappings.hashCode(), this)
  }
  fun add(str: String): Unit = str.stringAdd()
}
```

في المثال الثاني، ستُستدعَى دالة hashCode في نسخة Mappings.

### ث. 4.8.4 استبدال الدوال التابعة المُوسِّعة

يمكن إضـافة الكلمـة المفتاحيـة open إلى تعريـف الـدوال التابعـة المُوسّـعة إذا رغبت بالسـماح باسـتبدالها في الأصـناف الفرعيّة؛ وفي هـذه الحالـة، سـيكون النـوع المُسـتقبِل الموفـد وهميًا (virtual)، أي سـيكون نسـخةً وقت التشغيل (runtime instance)؛ ومع ذلـك، سيُسـتبيَن المسـتقبل المُوسَّـع (extension receiver) اسـتبيانًا ثابتًا دومًا:

```
open class Element(val name: String) {
  open fun Particle.react(name: String): Unit {
      println("$name is reacting with a particle")
  }
```

```
    open fun Electron.react(name: String): Unit {
        println("$name is reacting with an electron to make an  isotope")
    }

    fun react(particle: Particle): Unit {
        particle.react(name)
    }
}

class NobleGas(name: String) : Element(name) {
    override fun Particle.react(name: String): Unit {
        println("$name is noble, it doesn't react with particles")
    }

    override fun Electron.react(name: String): Unit {
        println("$name is noble, it doesn't react with electrons")
    }

    fun react(particle: Electron): Unit {
        particle.react(name)
    }
}
fun main(args: Array<String>) {
  val selenium = Element("Selenium")
  selenium.react(Particle())
  selenium.react(Electron())
  val neon = NobleGas("Neon")
  neon.react(Particle())
  neon.react(Electron())
}
```

مخرجات المثال السابق هي:

```
Selenium is reacting with a particle
Selenium is reacting with a particle
Neon is noble, and it doesn't react with particles
Neon is noble, and it doesn't react with electrons
```

يوضِّح هذا المثال كيف يعمل المستقبل مع الدوال المُوسَّعة المُستبدَلة. عرَّفنا زوجين من الأصناف؛ يتكون الزوج الأول من Element و NobleGas التي توسّع Element، ويتكون الزوج الثاني من Particle ونوعها الفرعي Electron.

نعرِّف في كلا الصنفيّن دالتين موسّعتين: الأولى في Particle والثانية في Electron.

يمكننا أن نرى من الناتج أنَّه لا يهم أي نوع من Particle أو Electron مررناه إلى الدالة react المعرَّفة في Element، فهي ستستدعي الدالة الملحقة المُعرَّفة في Particle، ويعود ذلك إلى تحديد نوع المستقبل تحديدًا ثابتًا، وهذا هو النوع الذي يُحدَّد عبر نوع المُصرِّف (compile type) وليس نوع وقت التشغيل (runtime type)، ويُعرِّف مدخل الدالة react لقبول النوع Particle، لذا هذا هو النوع التي نستخدمه لربط الدالة الملحقة.

عرَّفنا في NobleGas دالةً إضافيَّةً تقبل النوع الفرعي بشكل يستطيع المُصرِّف اختيار الدالة الأكثر تطابقًا، ويشبه هذا النوع من الإيفاد الثابت (static dispatch) التوابع الساكنة (static methods) في جافا.

## ج. توسيع الكائن المرافق

يمكن إضافة دوال موسّعة إلى كائنات مرافقة (Extension functions)، وستُستدعَى بعد ذلك مع الكائن المرافق بدلًا من نُسَخ الصنف.

وأبرز مثالٍ على متى يمكن أن يكون هذا مفيدًا هو إضافة دوال مُنتجة (factory functions) إلى النوع؛ فعلى سبيل المثال، قد نرغب بإضافة دالة إلى أعداد صحيحة لإرجاع قيمة عشوائيَّة مختلفة عند كل دعوة:

```
fun Int.Companion.random(): Int {
    val random = Random()
```

```
      return random.nextInt()
}
```

ثم يمكننا استدعاء الدالة الملحقة مثل المعتاد دون الحاجة إلى الكلمة المفتاحيَّة `companion`:

```
val int = Int.random()
```

هذا ليس مفيدًا مثل الدوال الملحقة العاديَّة، وهذا بسبب أنَّه يمكننا دائمًا إنشاء كائن جديد ووضع الدالة هناك أو إنشاء دالة في المستوى الأعلى، لكن قد ترغب في ربط دالة مع مجال اسم نوع آخر. كما في المثال السابق، من البداهة استدعاء الدالة `random()` مع النوع `Int` بدلًا من الدالة نفسها في صنفٍ باسم مثل `IntFactory` أو `RandomInts`.

## ح. إرجاع قيم مُتعدِّدة

لنفترض أنَّنا نريد حساب الجذور التربيعيَّة الموجبة والسالبة لعدد صحيح، يمكننا التعامل مع هذه المشكلة من خلال كتابة دالتيَّن مختلفتين مثل:

```
fun positiveRoot(k: Int): Double {
 require(k >= 0)
 return Math.sqrt(k.toDouble())
}

fun negativeRoot(k: Int): Double {
 require(k >= 0)
 return -Math.sqrt(k.toDouble())
}
```

ويمكن أيضًا إرجاع مصفوفة بطريقةٍ لا يتعيَّن علينا سوى استدعاء دالة واحدة:

```
fun roots(k: Int): Array<Double> {
   require(k >= 0)
   val root = Math.sqrt(k.toDouble())
   return arrayOf(root, -root)
```

```
}
```

ومع ذلك، لا نعرف من النوع المعاد إذا كان الجذر موجبًا أم سالبًا في الموقع 0، سنأمل في هذه الحالة أن يكون التوثيق صحيحًا، وإذا لم يكن كذلك، تفقَّد الشيفرة البرمجيّة، يمكننا تحسين هذا تحسينًا أفضل من خلال استخدام صنف مع خاصيتيّن تغلفان القيم المعادة:

```
class Roots(pos: Double, neg: Double)
fun roots2(k: Int): Roots {
   require(k >= 0)
   val root = Math.sqrt(k.toDouble())
   return Roots(root, -root)
}
```

لهذا ميّزة وجود حقول مسماة لذا يمكننا التأكد ما هو الجذر الموجب وما هو الجذر السالب، وكبديل للصنف المخصص هو استخدام نوع `Pair` من مكتبة كوتلن القياسيَّة، ويلف هذا النوع ببساطة قيمتين والتي يمكن الوصول إليها عبر الحقلين الأول والثاني:

```
fun roots3(k: Int): Pair<Double, Double> {
   require(k >= 0)
   val root = Math.sqrt(k.toDouble())
   return Pair(root, -root)
}
```

تُستخدَم هذه في الغالب عندما يكون من الواضح ما تعني كل قيمة؛ على سبيل المثال، لا تحتاج دالة ترجع رمز عملةٍ ومبلغٍ إلى صنفٍ مخصَّصٍ، إذ يمكن التفريق بينهما؛ وعلاوة على ذلك، إذا كانت الدالة محليَّةً، فقد تشعر أنَّ إنشاء صنفٍ مخصَّصٍ هو تكرارُ شيفرةٍ لشيء لن يكون مرئيًا خارج الدالة التابعة، وكما هو الحال دائمًا، كل حالة سيكون لها وضعًا مختلفًا.

**ملاحظة**

توجد نسخة لثلاث قيم من `Pair`، والتي اسمها `Triple`.

يمكننا تحسين ذلك أكثر باستخدام التصريح بالتفكيك (destructuring declaration) على الموقع المستدعي. يسمح التصريح بالتفكيك باستخراج القيم إلى متغيرات منفصلة تلقائيًا:

```
val (pos, neg) = roots3(16)
```

لاحظ المتغيرين الموجودين بين قوسين بعد الكلمة المفتاحية `val`، إذ ستُسنَد القيمة الأولية إلى الجذر الموجب، وستُسنَد القيمة الثانية إلى الجذر السالب، ويعمل هذا مع أي كائن يُنفِّذ واجهة مكونات خاصَّة.

يُنفِّذ النوع `Pair` المُضمَّن وجميع أصناف البيانات هذه الواجهة تلقائيًا. سنتحدث أكثر حول هذه الآليَّة في الفصل المخصَّص حول أصناف البيانات.

## خ. نمط التدوين الداخلي

يُغيِّر نمط التدوين الداخلي (Infix functions) من طريقة استدعاء الدالة، إذ يمكن باستعماله وضع معامل أو دالة بين العوامل (operands) أو الوسائط (arguments). ومثالٌ على ذلك في كوتلن هو الدالة `to`، والتي تُستخدَم لإنشاء نسخة من `Pair`:

```
val pair = "London" to "UK"
```

في كوتلن، يمكن استخدام الكلمة المفتاحية `infix` في تعريف الدوال التابعة، ويسمح لها هذا باستخدامها في النمط نفسه. ونظرًا لوضع دالة فُعِّل نمط التدوين الداخلي فيها بين معاملين، فيجب ألَّا يزيد عدد المعاملات المستعملة مع تلك الدوال عن معاملين اثنين فقط؛ المعامل الأول هو النسخة التي يفترض أن تُستدعَى الدالة معها، والمعامل الثاني هو المعامل الذي يفترض أن يُمرَّر صراحةً إلى الدالة في حال استدعيت وفق التدوين العادي.

إن أرت تعريف دالة يمكن استدعاؤها وفق التدوين الداخلي، استخدم الكلمة المفتاحية `infix` قبل الكلمة المفتاحية `fun`، وتذكَّر أنَّه لا يجب أن يزيد عدد المعاملات الصريحة المُمرَّرة إليها عن معامل واحد:

```
infix fun concat(other:String): String {
     return this + other
}
```

فعلى سبيل المثال، قد نريد إنشاء صنفٍ لحساب مصرفي قد يحتوي على رصيد، لذا قد نرغب بحوايته على

دالة تعمل على إضافة رصيدٍ لحساب العميل:

```
class Account {
     var balance = 0.0
     fun add(amount: Double): Unit {
            this.balance = balance + amount
     }
}
```

يمكننا استدعاء الدالة باستخدام صياغة النقطة العاديَّة:

```
val account = Account()
account.add(100.00)
```

أو يمكننا استدعاؤها بصياغة التدوين الداخلي، وذلك بإضافة الكلمة المفتاحية `infix` إلى تعريفها:

```
class InfixAccount {
     var balance = 0.0
     infix fun add(amount: Double): Unit {
            this.balance = balance + amount
     }
}
```

وبذلك يمكننا استدعاؤها بالشكل التالي:

```
val account2 = InfixAccount()
account2 add 100.00
```

في هذا المثال، كلا النمطين لا يؤثران على سهولة قراءة الشيفرة، لذلك من المرجح أن يستقر المرء على نمط النقطة العاديَّة. لكن يمكنك في بعض الأحيان الاستفادة نمط التدوين الداخلي.

وخير مثال على هذه الحالة هو الدوال قصيرة الاسم والمستخدمة بشكل متكرر مثل الدالة `to` الموجودة في مكتبة كوتلن القياسيَّة، وهذه الدالة هي دالة ملحقة لجميع الأنواع (مُعرَّفة في `Any`)، وتُستخدم لإنشاء نسخة من Pair، والذي هو مصرِّف بسيط لقيمتين.

رغم فائدة النوع `Pair`، إلا أنَّه يُحدِث عند إنشائه مباشرةً بعض الضوضاء في القيمتين. وازن بين السطرين المتشابهين من الشيفرة البرمجيَّة وأخبرني أيهما أسهل قراءةً:

```
val pair1 = Pair("london", "paris")
val pair2 = "london" to "paris"
```

السطر الثاني بالتأكيد هو الأقصر طولًا وأسهل قراءةً. وهذه الدالة بحد ذاتها مفيدةٌ للغاية عند ربط قيمة مع قيمة أخرى لإنشاء خريطة. وازن مجدّدًا بين نمطي الاستدعاء في السطرين التاليين:

```
val map1 = mapOf(Pair("London", "UK"), Pair("Bucharest", "Romania"))
val map2 = mapOf("London" to "UK", "Bucharest" to "Romania")
```

ومثال آخر للدوال المستدعاة بنمط التدوين الداخلي هي العمليات المجراة على البتات (اطلع على قسم الأنواع الأساسيَّة في الفصل الثاني، أساسيات كوتلن) والأجزاء المخصصة المكتوبة بلغات مخصَّصة المجال (DSLs).

إحدى اللغات مخصَّصة المجال التي تستفيد من التدوين الداخلي هي إطار الاختبار KotlinTest الذي يستخدمه لكتابة التوكيدات (assertions) في الاختبارات بالطريقة التي تتبعها اللغة الطبيعية. فعلى سبيل المثال، اطلع على المثال التالي:

```
myList should contain(x)
myString should startWith("foo")
```

سنغطي KotlinTest واختبارات DSL في الفصل الحادي عشر، الاختبار في كوتلن.

## 9. المعاملات

المعاملات (Operators) هي دوال تستخدم اسمًا رمزيًا، والعديد من المعاملات المدمجة في كوتلن هي استدعاءات دوال. فعلى سبيل المثال، الوصول إلى المصفوفة هو دالة حقيقيَّة:

```
val array = arrayOf(1, 2, 3)
val element = array[0]
```

في هذا المثال، تُرجمت العمليَّة [0] إلى استدعاء دالةٍ تدعى `get(index:Int)` المُعرَّفة في الصنف `Array`.

العديد من المعاملات مُعرَّفة مسبقًا في كوتلن كما في معظم اللغات الأخرى، وتميل معظمها إلى استعمال نمط التدوين الداخلي (infix style)، وهذا مألوف في مجموعة المعاملات الثنائيَّة التي تستعمل مع الأعداد.

| ملاحظة | رغم أنَّ كوتلن تعامل العمليات على الأنواع الأساسيَّة مثل معاملة الدوال، إلا أنَّها تُصرِّفها إلى عمليات بايتكود مقابلة لتجنُّب بطء الأداء الذي تتسبب به الدوال تحقيق أفضل أداء. |
|---|---|

يُفضَّل في كثير من الأحيان استعمال المعاملات بدلًا من أسمائها الحقيقيَّة إذا كانت المعاملات مألوفةً بالفعل إلى المستخدمين. لنأخذ مثلًا مجال الرياضيات أو الفيزياء، يشيع فيهما استخدام المعاملات شيوعًا كبيرًا، لذا لابد الاستمرار باستخدام تلك المعاملات الشائعة في لغات البرمجة. ففي مجال المصفوفات مثلًا، إنَّ استخدام المحرف + لجمع المصفوفات يبدو أكثر ألفةً من استخدام دالة باسم `add` أو `plus`، ومن السهل أيضًا قراءتها عند حذف الأقواس:

```
val m1: Matrix =
val m2: Matrix =
val m3 = m1 + m2
```

## أ. التحميل الزائد للمعاملات

تُعرَف القدرة على تعريف دوال تستخدم المعاملات «بالتحميل الزائد للمعاملات» (operator overloading). ويقصد بالتحميل الزائد للمعاملات أي تحميل معامل (رمز ثابت تعتمده لغة كوتلن مثل الرمز + والرمز *) بمجموعة معرفة مسبقًا من العمليات تُطبَّق على أنواع البيانات المختلفة.

بشكل عام، ستكون لغات البرمجة في مقياس بين عدم السماح بالتحميل الزائد للمعاملات وبين السماح باستخدام أي محرف تقريبًا؛ ففي جافا، تمنع اللغة من التعديل على المعاملات الدوال (operator functions) ولن يتمكن المطور من الإضافة عليها، لذلك تقبع جافا في أقصى الجانب الأيسر من هذا المقياس. ومن ناحية أخرى، فإن لغة سكالا متسامحة أكثر، وتسمح لك بالحصول على أي تركيبة تقريبًا، لذلك، فهي تقبع على الجانب الآخر من

المقياس.

لا يمكن القول بأفضلية أحد طرفي المقياس على الآخر، إذ يعتمد ذاك على وجهة نظرك. فعدم السماح بالتحميل الزائد للمعاملات يعني أنه لن يتمكن المطورون من إساءة استعمال المعاملات بِعدّها أسماء دوال؛ ومن جانب آخر، فإن السماح بهذا يعني إنشاء لغات مخصصة المجال (DSL، اختصارًا للعبارة Domain-Specific Language) قويّة تعالج مشاكل مُحدَّدة.

اختار مصممو كوتلن حلًا وسطًا وذلك بالسماح بالتحمل الزائد للمعاملات بطريقة ثابتة ومراقبة، وهنالك قائمة ثابتة من المعاملات التي يمكن استخدامها على أنَّها دوال، ولكن يُحظر استخدام أي معاملات أخرى عشوائية. ولإنشاء مثل هذه الدالة، يجب وضع الكلمة المفتاحية `operator` قبلها ويجب تعريف الاسم الأجنبي المكافئ للمعامل أيضًا.

تملك جميع المعاملات اسمًا أجنبيًا مكافئًا مُحدَّد مسبقًا يُستخدم في عملية التحميل الزائد للمعامل، ويعيد المُصرِّف كتابة استخدام المعامل لاستدعاءات الدالة.

| ملاحظة | يمكن تعريف المعاملات على أنَّها دوال تابعة (member functions) أو دوال مُوسَّعة (extension functions). |
|---|---|

سنعيد استخدام المثال السابق لجمع المصفوفات، وذلك للاستفادة من التحميل الزائد للمعاملات بالشكل التالي:

```
class Matrix(val a: Int, val b: Int, val c: Int, val d: Int) {
     operator fun plus(matrix: Matrix): Matrix {
             return Matrix(a + matrix.a, b + matrix.b, c + matrix.c, d +
matrix.d)
     }
}
```

هذه حالة بسيطة تسمح بمصفوفتين فقط.

عرَّفنا دالة باسم `plus` تجمع بين مصفوفتين؛ لاحظ كيف ميَّزنا الدالة باستخدام الكلمة المفتاحية `operator`

قبل الكلمة المفتاحية fun؛ وكما ذكرنا في الفصل السابق، يجب تمييز الوسائط باستخدام val لاستخدامها داخل الدوال التابعة.

يمكننا تنفيذ شيفرة هذا الصنف بالطريقة التالية:

```
val m1 = Matrix(1, 2, 3, 4)
val m2 = Matrix(5, 6, 7, 8)
val m3 = m1 + m2
```

تُصرَّف هذه الشيفرة إلى الشيفرة التالية المقابلة لها:

```
val m1 = Matrix(1, 2, 3, 4)
val m2 = Matrix(5, 6, 7, 8)
val m3 = m1.plus(m2)
```

رغم أنَّ هذا المثال بسيط، إلا أنه يوضِّح مدى سهولة استخدام التحميل الزائد على المعاملات.

يمكن استدعاء الدالة أيضًا باستخدام نمط النقطة العادي إذا لزم الأمر، إذ سيُستخدَم على أي حال اسم الدالة الحقيقي بدلًا من رمز المعامل. ورغم أنَّ هذا لا يكسبك أي فائدة تُذكَر في هذا المثال، إلا أنَّ هنالك حالات يساعدك فيها هذا الأمر على جني فائدة عظيمة.

لا تقتصر المعاملات الدوال على العمل على نوع الصنف نفسه المُعرَّفة ضمنه، فبإمكانك مثلًا تعريف الصنف List وإضافة عناصر للقائمة باستخدام المعامل + وحذف عناصر منها باستخدام المعامل -.

### ب. المعاملات الأساسيَّة

ذكرنا مسبقًا أن كوتلن اتخذت حلًا وسطًا في موضوع التحميل الزائد للمعاملات وحدَّدت قائمة أساسية وثابتة بتلك المعاملات التي تقبل تحميلًا زائدًا، وإليك قائمة المعاملات الأساسيَّة هذه وأسماء الدوال المقابلة لها:

| العمليَّة | اسم الدالة |
|---|---|
| a + b | a.plus(b) |
| a - b | a.minus(b) |
| a * b | a.times(b) |

| a / b | a.div(b) |
|---|---|
| a & b | a.mod(b) |
| a..b | a.rangeTo(b) |
| +a | a.unaryPlus() |
| -a | a.unaryMinus() |
| !a | a.not() |

تدعم كوتلن معاملات أخرى بالإضافة إلى هذه القائمة.

### ت. الكلمة المفتاحية in والدالة contains

يمكنك تحميل الكلمة المفتاحية `in` تحميلًا زائدًا في صنفك، والتي تعرَّفت عليه مسبقًا من حلقات `for` أو من التحقُّق من التجميعات (collections)؛ ويكون الاسم المقابل لها هو `contains`. وإليك أمثلة عن كيفيَّة استخدامهما:

```
val ints = arrayOf(1,2,3,4)

val a = 3 in ints
val b = ints.contains(3)

val c = 5 !in ints
val d = !ints.contains(5)
```

### ث. الجالب والضابط (get/set)

الدالتان المقابلتان لعمليتي جلب قيمة من مصفوفة وضبطها عبر صياغة الأقواس المعقوفة هما `get` و `set`، ويعدُّ عدد الوسائط المُمرَّرة إلى `get` و `set` بالترتيب عشوائيًا. وإليك مثال عن عمل الأقواس المعقوفة في أصناف مثل `list` و `collection` للوصول إلى قيمةٍ محدَّدةٍ وجلبها:

```
private val list = listOf(1, 2, 3, 4)
val head = list[0]
```

المثال التالي يستخدم `get` و `set` مع أكثر من وسيط موضعي (position argument):

```
enum class Piece {
   Empty, Pawn, Bishop, Knight, Rook, Queen, King
}

class ChessBoard() {
   private val board = Array<Piece>(64, { Piece.Empty })
   operator fun get(rank: Int, file: Int): Piece = board[file * 8 +  rank]

   operator fun set(rank: Int, file: Int, value: Piece): Unit {
      board[file * 8 + rank] = value
   }
}
```

عرَّفنا هنا صنفًا يمثِّل مربعات رقعة الشطرنج، إذ عرَّفنا اللوحة على أنَّها مصفوفة مؤلفة من 64 عنصر (مربع)، وكل عنصر فارغ مبدئيًا. يمكننا جلب get أو ضبط set أي عنصر موجود في موضع محدَّد باستخدام الإحداثيات التي تمثِّل صف وعمود رقعة الشطرنج:

```
val board = ChessBoard()
board[0, 4] = Piece.Queen
println(board[0, 4])
```

### الاستدعاء (invoke)

يمكن استخدام الأقواس على أنَّها معاملات أيضًا عن طريق تسمية دالة بالاسم invoke. وفي هذه الحالة، نستدعي الدالة مباشرةً على النسخة، إذ يجعل هذا الصنف نفسه يبدو وكأنَّه دالة:

```
class RandomLongs(seed: Long) {
      private val random = Random(seed)
      operator fun invoke(): Long = random.nextLong()
}
```

غلَّفنا في هذا المثال Random مع بذرة مُخصَّصة ثمَّ سمحنا للمستخدم باستدعاء الصنف مباشرةً لتوفير الاستخدام التالي:

```
fun newSeed(): Long = /// some secure seed
val random = RandomLongs(newSeed())
val longs = listOf(random(), random(), random())
```

لا توجد قيود على عدد دوال `invoke`، ويمكنك تحميلها تحميلًا زائدًا عبر النوع وعدد المعاملات:

```
object Min {
   operator fun invoke(a: Int, b: Int): Int = if (a <= b) a else b
   operator fun invoke(a: Int, b: Int, c: Int): Int = invoke(invoke(a, b), c)
   operator fun invoke(a: Int, b: Int, c: Int, d: Int): Int = invoke(invoke(a, b), invoke(c, d))

   operator fun invoke(a: Long, b: Long): Long = if (a <= b) a else b
   operator fun invoke(a: Long, b: Long, c: Long): Long = invoke(invoke(a, b), c)
   operator fun invoke(a: Long, b: Long, c: Long, d: Long): Long = invoke(invoke(a, b), invoke(c, d))
}
```

في هذا المثال، أنشأنا إصدارات متعددةً من `min`، وستستدعي كوتلن الإصدار الذي يتوافق مع المعاملات المعطاة؛ ويمكن استدعاؤها على النحو التالي:

```
min(1, 2, 3)
min(1L, 2L)
```

### ج. الموازنة

معاملات الموازنة، أصغر من وأكبر من وأصغر أو يساوي وأكبر أو يساوي، جميعها قابلة للتحميل الزائد، وتتطلّب كل هذه المعاملات الأربع دالةً واحدةً مشتركة بينها تسمى `compareTo`. يجب أن تُرجع هذه الدالة عددًا صحيحًا (Int) وأن تكون متناسقة مع الواجهة `Comparator` في جافا، ولذلك يجب أن تعيد عددًا صحيحًا سالبًا للإشارة إلى أنَّ a أصغر من b، وعددًا صحيحًا موجبًا عندما يكون b أكبر من a، والعدد 0 عند تساوي a مع b:

```kotlin
class BingoNumber(val name: String, val age: Int) {
   operator fun compareTo(other: BingoNumber): Int {
      return when {
         age < other.age -> -1
         age > other.age -> 1
         else -> 0
      }
   }
}
```

عرّفنا هنا BingoNumber وهو رقم الكرة ولقب يجب على مستدعي Bingo الهتاف له، ويمكننا الآن باستخدام دالة compareTo موازنة أرقام Bingo باستخدام < و > و <= و >=:

```kotlin
val a = BingoNumber("Key to the Door", 21)
val b = BingoNumber("Jump and Jive", 35)
println(a < b) // true
println(b < a) // false
```

## ح. الإسناد

تدعم كوتلن التحميل الزائد لمعاملات الإسناد المختصرة (المركبَّة) مثل += للمتغيِّرات القابلة للتغيير، ويمكنك استخدام معاملات الإسناد القياسيَّة أيضًا؛ فعلى سبيل المثال، يعمل التعريف التالي لكل منهما، القياسية والمركبَّة:

```kotlin
class Counter(val k: Int) {
   operator fun plus(j: Int): Counter = Counter(k + j)
}
var counter = Counter(1)
counter = counter + 3
counter += 2
```

ومع ذلك، إذا كنت تريد السماح بعمليات إسناد ليست ضمن العمليات الأساسيَّة، يمكنك فعل ذلك باستخدام

الشيفرة التالية:

```
class Counter(var k: Int) {
   operator fun plusAssign(j: Int): Unit {
      k += j
   }
}
var counter = Counter(1)
counter += 2
```

في هذه الحالة، يجب على الدالة أن تُرجع `Unit`.

**تنبيه**

لا يمكنك تعريف كلا نوعي العمليات، بل تستطيع تعريف إمَّا `plus` أو `plusAssign` مثلًا وليس كليهم معًا؛ ويمكن أن يُستخدَم الأوَّل للإسناد وغير الإسناد دومًا، ولكن يستخدم الأخير للإسناد فقط.

يحتوي الجدول التالي على أسماء التوابع المقابلة للمعاملات:

| العمليَّة | اسم الدالة |
| --- | --- |
| `a += b` | `a.plusAssign(b)` |
| `a -= b` | `a.minusAssign(b)` |
| `a *= b` | `a.timesAssign(b)` |
| `a /= b` | `a.divAssign(b)` |
| `a %= b` | `a.modAssign(b)` |

ربما قد لاحظت التشابه بين هذا والعمليات الأساسيَّة، إذ أضفنا الكلمة `Assign` إلى نهاية الاسم فقط.

### خ. التشغيل المتوافق مع جافا

لا تدعم جافا خاصية التحميل الزائد للمعاملات الموجودة في كوتلن، لذلك لا يوجد ما يعادل الكلمة المفتاحية `operator` في جافا؛ وللتغلّب على هذا، تسمح كوتلن أي تابع جافا يملك بصمةً صحيحةً (correct signature) باستخدامه على أنَّه مُعامل.

فعلى السبيل المثال، خذ بصمة تابع جافا هذا المُعرَّف في صنف يُسمَى `Matrix`، والذي يُطابق الاسم المقابل للمعامل +:

```
public Matrix plus(Matrix other) { }
```

يمكن استدعاء هذا في كوتلن على النحو التالي:

```
val matrix3 = matrix1 + matrix2
```

## 10. الصياغة المختصرة للدوال (الدوال المجرَّدة)

يمكننا تعريف «دالة مجرَّدة» (function literal، ويقال لها «دالة مجهولة» أيضًا[7]) تمامًا مثلما عرَّفنا سلسلة نصية باستعمال علامتي الاقتباس مباشرةً،"hello"، أو عدد عشري بكتابته مباشرةً، 12.34؛ وللقيام بذلك، نُغلِّف الشيفرة بقوسين معقوصين (كما أخبرتك في الهامش، فهذا هو الرمز المستعمل لتعريف دالة مجهولة أو مجرَّدة function literal بطريقة سريعة مختصرة) مثل:

---

7 انتبه إلى أنَّه لا توجد ترجمة معتمدة للمصطلح function literal (على الأقل على مستوى موسوعة حسوب)، واعتمدت هذه الترجمة، دالة مجرَّدة أو مجهولة، في هذا الكتاب نظرًا لأنَّها الأقرب وليست الأصح أو الأنسب، إذ يجب عليك فهم ماهية الكلمة literal والتي تعني تعريف نوعٍ تعريفًا مختصرًا بناءً على صياغة ورموز يجري اتباعها في اللغة بدلًا من استدعاء بانٍ لبناء النوع في كل مرَّة نريد فيها أن ننشئ قيمة من ذلك النوع، فبدلًا من استدعاء الباني `String()` لبناء سلسلة نصية، يجري استعمال علامتي الاقتباس `""` لبناء سلسلة نصية بطريقة سريعة ومختصرة والأمر مماثل للدوال وبقية الأنواع.

```
{ println("I am a function literal") }
```

يمكن إسناد دالة مجرّدة إلى متغيّرة مثل القيم المجرّدة الأخرى:

```
val printHello = { println("hello") }
printHello()
```

لاحظ في هذا المثال أنَّه عند تعريف دالة مجرَّدة، يمكننا استدعاؤها لاحقًا باستخدام الأقواس كما نستدعي الدالة العاديَّة، وبالطبع بمجرَّد تعريفها، يمكننا استدعاؤها عدة مرات.

يمكن للدالة المجرَّدة المجهولة أن تقبل معاملات، ولهذا نكتب المعاملات مع الأنواع قبل السهم الصغير الذي يدل على جسم الدالة:

```
val printMessage = { message: String -> println(message) }
printMessage("hello")
printMessage("world")
```

كما ترى، نمرِّر المعاملات عند الاستدعاء كما في الدالة العاديَّة، ويمكننا تجنب التصريح عن نوع المعامل عند استخدام الدالة المجرَّدة في مكان حيث يمكن للمصرِّف أن يتعرف على أنواع المعاملات:

```
{ message -> println(message) }
```

في الحقيقة، تملك كوتلن خدعة أفضل، فإذا كان هنالك معامل واحد ويمكن للمصرِّف أن يخمِّن نوعه، فيسمح حينئذٍ لنا بحذف المعامل تمامًا، ويمكن آنذاك استخدام المتغيِّر الضمني `it`:

```
{ println(it) }
```

إنَّ استخدام الدالة المجرَّدة هي أساس الدوال ذات المرتبة الأعلى (higher order function)، وسنرى هذا في الفصل القادم. وتُستخدَم الدوال المجرَّة أيضًا هذه في التوابع المجرَّدة الوحيدة (Single Abstract Methods)، التي سنُغطيها لاحقًا في هذا الفصل.

## 11. الدوال التعاودية

التعـاود (Recursion) هـو أداة برمجيَّة قويَّة يسـتخدمها أغلب المـبرمجين، والدالـة التعاوديـة هي دالـةٌ تستدعي نفسها عند تحقُّق شروط معيَّنة، وخير مثال شائع على ذلك هو متتالية فيبوناتشي وهي: القيمـة التاليَّة هي مجموع القيمتين السابقتين؛ ويمكننا كتابة متتالية فيبوناتشي برمجيًا بالشكل التالي:

```
fun fib(k: Int): Int = when (k) {
     0 -> 1
     1 -> 1
     else -> fib(k - 1) + fib(k - 2)
}
```

لاحظ أننا عرَّفنا حالات أساسيَّة للمعامل k وهي 0 و 1 والتي لا تستخدم الاسـتدعاء الـذاتي، أمَّا من أجـل القيم الأكبر، تَستدعِي الدالة نفسها مع القيمتين السابقتيّن لقيمة k، وهكذا.

هذه الشيفرة مختصرة للغاية، لكنها ليسـت الأكـثر كفـاءة، ففي كـل مـرة تسـتدعي الدالـة fib نفسـها، يجب أن يحتفـظ المُشـغِّل الآني بسلسـلة اسـتدعاءات الدالة تلـك في المكدس لتُسـتأنَف تنفيـذها مـتى مـا انتهي تنفيـذ آخـر استدعاء وأعاد قيمةً (القيمة 1 في حالتنا عندما تصبح قيمـة k هي 1 أو 0)، ويمكننـا توضيـح هـذا عن طريـق هـذه الشيفرة الزائفة (Pseudocode) التاليّة:

```
invoke fib with k:
     If k == 0 then return 1
     If k == 1 then return 1
     Let temp1 = invoke fib with k-1
     Let temp2 = invoke fib with k -2
     return temp1 + temp2
```

كما ترى، بعد اكتمال الاستدعاءات التعاودية، نضـيفهم معًا؛ ولـذلك، يُبقِي المصـرّف هـذا المكـدس حيًا لتخـزين المتغير temp1 والمتغير temp2، وإذا استُدعيَت الدالـة fib عـددًا كبـيرًا من المـرات، فسـيكون عـدد الاسـتدعاءات التعاودية المطلوبة كبيرًا قبل الوصول إلى الاستدعاء الأخير ولذلك يجب الانتباه من نفاد مساحة المكدس الـذي قـد يؤدي إلى خطأ «طفحان المكدس» (stack overflow) الشهير.

**تلميح**

يمكن تحسين هذه الدالة عن طريق تذكُّر قيم `fib(k-1)` بدلًا من إعادة حسابها كل مرة، ومع ذلك، فإن هذا المثال يوضِّح لك كيف يُحدِث الاستدعاء التعاودي معضلةً إذا كان عدد الاستدعاءات غير محدود.

إذا كان استدعاء دالة تعاودية هو آخر عمليّة استدعاء في دالة معيّنة وكان نتيجة هذا الاستدعاء هو ببساطة إرجاع قيمة، لن يحتاج النظام إلى الاحتفاظ بإطار الاستدعاء السابق في المكدس، لأنه لن يحتاج إلى متغيرات أخرى للعمليات، وسيُرجع ببساطة قيمةً من الاستدعاء التعاودي، وتُسمَى هذه التقنيّة «بذيل الاستدعاء التعاودي» (tail recursion)، وهي تسمح لنا بكتابة خوارزميات استدعاء تعاودي فعّالة تجنبنا الوقوع في خطأ طفحان المكدس.

لإبلاغ كوتلن أن دالتنا يُتوقَّع أن تكون دالة تعاودية تذييلية (tail recursive function)، نستخدم الكلمة المفتاحيّة `tailrec` عند تعريف الدالة، ومن ثم سيضمن المصرّف أن كل استخدام للاستدعاء التعاودي في الدالة هو آخر عمليّة، وإذا لم يكن كذلك، فسيحدث خطأ وقت التشغيل.

إليك مثال لدالة حساب مضروب عدد (factorial) بطريقة الاستدعاء التعاودي، ولاحظ أن مضروب العدد 0 (أي !0) هو 1:

```
fun fact(k: Int): Int {
     if (k == 0) return 1
     else return k * fact(k - 1)
}
```

العمليّة الأخيرة ليست استدعاءً تعاوديًا لأنَّ نتيجة الاستدعاء التعاودي تُضرَب قبل إرجاعها، ومع ذلك، فإذا أعدنا كتابة الدالة لتحمل النتيجة معها، فيمكننا الرجوع مباشرة من الاستدعاء التعاودي:

```
fun fact(k: Int): Int {
     fun factTail(m: Int, n: Int): Int {
          if (m == 0) return n
          else return factTail(m - 1, m * n)
```

```
    }
    return factTail(k, 1)
}
```

الدالة factTail الداخليّة الآن هي ذيل الاستدعاء التعاودي، ويمكننا وضع علامة عليه حتى يؤكد لنا المصرّف ذلك:

```
fun fact(k: Int): Int {
    tailrec fun factTail(m: Int, n: Int): Int {
        if (m == 0) return n
        else return factTail(m - 1, m * n)
    }
    return factTail(k, 1)
}
```

## 12. عدد متغيِّر من الوسائط

تسمح كوتلن بتعريف دوالٍ تقبل عددًا متغيّرًا من الوسائط، وتسمى هذه الميّزة بالاسم varargs، اختصارًا للعبارة variable number of arguments، وهي تسمح للمستخدمين بالتمرير قائمة من المعاملات مفصولة بفاصلة، والتي سوف يجمعها المصرّف تلقائيًا في مصفوفة. ويعرف مطورو جافا هذه الميّزة تمام المعرفة، والتي تبدو في جافا كالتالي:

```
public void println(String.. args) { }
```

ويقابل ذلك في كوتلن استخدام الكلمة المفتاحية vararg قبل اسم معامل:

```
fun multiprint(vararg strings: String): Unit {
    for (string in strings)
    println(string)
}
```

ويمكنك استدعاء الدالة multiprint مع أي تمرير أي عدد تريد من الوسائط إليها:

```
multiprint("a", "b", "c")
```

يمكن أن تحتوي الدوال على معاملات عاديّة وعلى معامل vararg واحد على الأكثر:

```
fun multiprint(prefix: String, vararg strings: String): Unit {
     println(prefix)
     for (string in strings)
            println(string)
}
```

يُفضَّل أن يكون المعامل vararg آخر المعاملات، ولكن لا يجب اتباع ذلك بالضرورة، فإذا كان هنالك معـاملات أخـــــــرى بعـــــــد vararg، فيجب تمريـــــــــر المعـــــــــاملات بأســــــــمائها أي باســــــــتخدام المعاملات المسماة:

```
fun multiprint(prefix: String, vararg strings: String, suffix: String):
Unit {
   println(prefix)
   for (string in strings)
     println(string)
   println(suffix)
}
```

حُصِر المعامل vararg بين المعامل المسمى prefix والمعامل المسمى suffix وتمرَّر المعاملات آنذاك بالشكل التالي:

```
multiprint("Start", "a", "b", "c", suffix = "End")
```

## أ. معامل النشر

إذا عُرِّفَت دالةٌ تقبل عددًا متغيرًا من المعاملات باستخدام vararg، وكان لديك مصفوفة بالفعل، فكيف تُمرِّرها؟ جواب ذلك في استخدام معامل النشر (spread operator) وهو *، الذي ينشر عناصر المصفوفة ويمرِّرها على أنَّها معاملات فرديَّة.

افترض أنَّ لدينا مصفوفة من سلاسل نصيَّة نريد تمريرها إلى دالة تدعى multiprint عرَّفناها سابقًا،

فستكون الشيفرة البرمجيَّة كالتالي:

```
val strings = arrayOf("a", "b", "c", "d", "e")
multiprint("Start", *strings, suffix = "End")
```

لاحظ أنّنا أرفقنا معامل النشر في بداية اسم المُتغيِّر.

| تنبيه | في إصدارات كوتلن الحاليَّة، يعمل معامل النشر مع المصفوفات فقط، وسيُرفَع هذا القيد في الإصدارات المستقبليَّة. |
|---|---|

## 13. دوال المكتبة القياسيَّة

توفّر كوتلن مكتبةً قياسيَّةً تكمّل مكتبة جافا القياسيَّة ولا تستبدلها، فهنالك دوال عديدة تعمل على جعل أنواع جافا وتوابعها ملائمةً للعمل في بيئة كوتلن. سنغطي في هذا الفصل بعض الدوال منخفضة المستوى (lower level function) التي سنستخدمها على المدى البعيد.

### أ. apply

تُعدُّ `apply` إحدى الدوال الموسّعة في مكتبة كوتلن القياسيَّة المُعرَّفة في `Any`، لذلك يمكن استدعاؤها على النُسَخ المشتقة من جميع الأنواع. تقبل `apply` استدعاء تعبير لامدا (lambda) مع المستقبل الذي يمثِّل النسخة الذي تُستدعَى معه، وسترجِع الدالة `apply` بعدئذٍ النسخة الأصلية.

يكمن الاستخدام الأساسي لهذه الدالة في جعل الشيفرة البرمجيَّة التي تحتاج إلى تهيئة نسخة سهلة القراءة عن طريق السماح باستدعاء الدوال والخاصيات داخل الدالة مباشرةً قبل إرجاع القيمة نفسها، كما في المثال التالي:

```
val task = Runnable { println("Running") }
Thread(task).apply { setDaemon(true) }.start()
```

أنشأنا هنا المهمة `task` التي هي نسخة من `Runnable` ثمَّ أنشأنا نسخة خيط جديد (نسخة من Thread) لتشغيل هذه المهمة، وضبطنا في داخل المُغلِّف (closure)، نسخة الخيط حتى يكون خيط خفي (daemon thread).

لاحظ أنَّ الشيفرة البرمجيـة للمُغلِّف تعمل على نسـخة الخيـط مباشـرةً، والنسـخة الـتي اسـتدعينا معها الدالـة apply هي مُستقبِل المُغلِّف. ولاحظ كـذلك، أنـه يمكننـا اسـتدعاء start() مـع القيمـة المعـادة لأنَّ الدالـة apply ترجع النسخة الأصلية دائمًا، بغض النظر عن ما سيُرجعه المُغلِّف نفسه.

إن لم نستخدم الدالة apply، فستبدو الشيفرة بالشكل التالي:

```
val task = Runnable { println("Running") }
val thread = Thread(task)
thread.setDaemon(true)
thread.start()
```

## ب. let

الدالة let هي دالةٌ مُوسّعةٌ في مكتبة كوتلن القياسـيّة وتشـبه الدالـة apply إلى حـدٍ مـا، إذ الفـرق الجـوهري بينهما هو أنها تُرجع قيمة المُغلِّف نفسه؛ وهي مفيدة عندما ترغب في تنفيذ بعض التعليمات البرمجيَّة على كائن قبـل إرجاع قيمة مختلفة ولا تحتاج إلى إبقاء مرجع للكائن الأصلي:

```
val outputPath = Paths.get("/user/home").let {
      val path = it.resolve("output")
      path.toFile().createNewFile()
      path
}
```

لاحظ أنَّ it تشير إلى النسخة الـذي اسـتدعينا run عليهـا، وهي مجلـد home للمسـتخدم. والفائـدة من كتابـة الشيفرات البرمجيَّة بهذه الطريقة هو أنَّنا لا نحتاج إلى إسناد المسار الأصلي إلى متغيِّر وسيط.

## ت. with

الدالة with دالةٌ الأعلى مرتبـةً (top-level function) ومُصـمَّمة للحـالات الـتي تريـد فيهـا اسـتدعاء دوال مختلفة على كائن ولا ترغب في تكرار ذكر مُستقبِل كـل مـرة. تقبـل الدالـة مسـتقبلًا ومُغلِّفًـا تريـد أن تنفِّذه منـع ذلـك المستقبل:

```
val g2: Graphics2D = ...
g2.stroke = BasicStroke(10F)
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON)g2.setRenderingHint(Renderi
ngHints.KEY_DITHERING,  RenderingHints.VALUE_DITHER_ENABLE)
g2.background = Color.BLACK

with(g2) {
   stroke = BasicStroke(10F)
   setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON)
setRenderingHint(RenderingHints.KEY_DITHERING,
RenderingHints.VALUE_DITHER_ENABLE)
   background = Color.BLACK
}
```

في هذا المثال، تعمل المجموعة الأول من الاستدعاءات على المرجع `g2` مباشرةً، وفي المجموعة الثانية، عُيِّن المستقبل إلى `g2`، لذلك يمكن استدعاء الدوال عليه مباشرةً..

### ث. run

الدالة `run` دالةٌ موسّعةٌ تجمع بين حالتي استخدام الدالة `with` والدالة `let`، أي عند تمرير مُغلّف لها، والذي يملك نسخةً تمثّل المستقبل، ستعيد الدالة `run` القيمة التي يعيدها المُغلّف نفسه:

```
val outputPath = Paths.get("/user/home").let {
      val path = resolve("output")
      path.toFile().createNewFile()
      path
}
```

الفارق الجوهري بين الدالة `let` والدالة `run` هو أن المستقبل في الأخيرة هو النسخة بينما يكون وسيط المُغلّف في الأولى هو النسخة.

### ج. lazy

الدالـة `lazy` هي إحـدى الـدوال المفيـدة أيضًـا الـتي تُغلّف اسـتدعاء دالـة مسـتنزف للأداء أو الـذاكرة أو الموارد ...إلخ. لتُستدعَى عند الحاجة إليها:

```
fun readStringFromDatabase(): String = ... // عملية مُستنزِفة
val lazyString = lazy { readStringFromDatabase() }
```

في المرة الأولى التي نطلب فيها النتيجة، يمكننا الوصول إلى القيمة من مرجع تولّده الدالة `lazy` (انظر المثال أدناه)، وحينها فقط ستُستدعَى الدالة المُغلّفة فعليًا:

```
val string = lazyString.value
```

هذا النمط شائع في الكثير من لغات البرمجة والإطارات، وفائدة اسـتخدام هـذه الدالـة المدمجـة بـدلًا من فعـل ذلك بنفسك هو أن التزامن قد حُسِب حسـابه من أجلـك؛ ولهـذا، إذا طُلبَت القيمـة مـرّتين، فسـتعالج كـوتلن بأمـان أي حالات تسابق (race conditions) عن طريق تنفيذ الدالة الضمنيَّة مرةً واحدةً.

### ح. use

تشبه الدالة `use` التعليمة `try-with-resources` الموجودة في جافا 7، فقد عُرِّفَت على أنَّها دالـة مُوسّـعة لنسخة موردٍ قابل للإغلاق (closeable resource)، وتقبل دالةً مجهولةً (مجرَّدةً) تجري عمليةً على هذا المورد، وستستدعي `use` تلك الدالة بأمان، وستُغلِق المورد بعد انتهاء الدالة سواء رُفِع استثناء أم لا:

```
val input = Files.newInputStream(Paths.get("input.txt"))
val byte = input.use({ input.read() })
```

عمومًا، تعدُّ الدالة `use` الوسيلة الأكثر إيجازًا في التعامـل مع المـوارد في الحـالات البسـيطة، دون الحاجـة إلى كتلة `try/catch/finally`.

### خ. repeat

تكرِّر الدالة `repeat` – كما يوحي اسمها – تنفيذ دالة مجهولة (مجرَّدة) عددًا محـدَّدًا من المـرات، إذ تقبـل دالـة

مجهولة وعدد صحيح، وتُستخدَم هذه الدالة في الحالات البسيطة التي تحتاج فيها إلى تكرار تنفيذ شيءٍ ما دون اللجوء إلى كتل تكرارية (مثل for):

```
repeat(10, { println("Hello") })
```

### د. require/assert/check

تُوفِّر لنا كوتلن ثلاثة دوال تمكّننا من إضافة قدرٍ محدّدٍ من «المواصفات الاصطلاحية» (formal specifications) لبرنامجنا والتي هي توكيدات إمّا أن تكون محقَّقة (أي تحمل القيمة true) أو غير محقَّقة (أي false) حيث يجري التحقق منها؛ ويشار إليها باسم «العقود» (contracts) أو «تصميم حسب العقد» (design by contract):

- ترمي الدالة require استثناءً ويُستخدَم للتأكُّد من أنَّ الوسائط تتطابق مع شروط المدخلات.
- ترمي الدالة assert استثناءً من النوع AssertionException ويُستخدَم لضمان اتساق الحالة الداخليّة (internal state).
- ترمي الدالة check استثناءً من النوع IllegalStateException ويُستخدَم أيضًا لاتساق الحالة الداخليّة.

تتشابه هذه الدوال كثيرًا، والفرق الرئيسي بينها هو في نوع الاستثناء الذي ترميه. يمكن تعطيل عمل الدالة assert وقت التشغيل، ولكن لا يمكن تعطيل عمل الدالة require والدالة check، وهذا مثال على ذلك:

```
fun neverEmpty(str: String) {
     require(str.length > 0, { "String should not be empty" })
     println(str)
}
```

في هذا المثال، نضمن دائمًا عدم تمرير سلسلة نصيّة فارغة، إذ تُقيِّم الدالة المُجرَّدة التي مُرِّرت على أنَّها رسالة إلى الدالة require بتكاسل (lazily evaluated)، ولن تُستدعَى إذا كان الشرط محقَّقًا (أي true).

## 14. الدوال المُعمَّمة

هل سبق لك أن كتبت دالة لأحد الأنواع ثم احتجت إلى كتابتها مرةً أخرى لنوع آخر؟ ربما كتبت دالة تعمل مع السلاسل النصيّة ثم اضطررت لكتابة نفس الدالة مرة أخرى للعمل مع الأعداد الصحيحة.

لتجنب هذه الحالة، يمكن تعميم الدوال في الأنواع التي تستخدمها، ويسمح هذا للدالة بأن تُكتَب بطريقةٍ محدَّدةٍ لتعمل مع أي نوع بدلًا من تقييد عملها مع نوع محدّد فقط؛ ولفعل هذا، يجب تعريف نوع المعاملات في توقيع الدالة بالشكل التالي:

```
fun <T> printRepeated(t: T, k: Int): Unit {
     for (x in 0..k) {
            println(t)
     }
}
```

يعمل هذا المثال على طباعة العنصر t عدد k من المرات، وقد تفكر أنه كان بإمكاننا تعريف هذه الدالة باستخدام النوع Any وستعمل، بما أن الدالة println يمكنها قبول النوع Any نفسه، وهذا صحيح؛ ومع ذلك، ما لا يمكنك القيام به مع النوع Any هو التأكد أنَّ المعاملات المُمرَّرة تنتمي إلى نوع واحد وأنَّ نوع القيمة المراد إعادتها هو نفس نوع القيمة المعطاة (المُمرَّرة). دعنا نفترض أنَّنا نريد دالة تُرجع عنصرًا عشوائيًا من أصل ثلاث عناصر تُمرَّر إليها:

```
fun <T> choose(t1: T, t2: T, t3: T): T {
     return when (Random().nextInt(3)) {
            0 -> t1
            1 -> t2
            else -> t3
     }
}
```

والآن، سيفرض المُصرِّف عند استدعاء هذه الدالة أن تكون العناصر الثلاثة من نفس النوع، ولا يهم أي نوع طالما أنها من نوع واحد؛ أضف إلى ذلك أنَّ نوع القيمة المعادة سيكون من نفس نوع القيم المعطاة أيضًا، لذلك

ستعمل الشيفرة التالية مع المُتغيِّر r وسيُستنتَج النوع من القيم المعطاة التي هي أعداد صحيحة (أي سيكون هنا Int):

```
val r = choose(5, 7, 9)
```

إذا لم توفر كوتلن دوالًا مُعمَّمةً (generic functions)، فسنضطر إلى كتابة دالة منفصلة لكل نوع مختلف نرغب في استخدامه أو كتابة دالة تُرجع قيمة من النوع Any، وسيكون عيب استخدام Any هنا هو أن نوع المخرجات يجب أن يكون Any أيضًا، وبالتالي يجب على المستدعي اللجوء إلى التحويل بين الأنواع (casting) للعودة إلى النوع الأصلي.

لا تقتصر الدوال على نوع مُعمَّم واحد، ويمكنها أيضًا تحديد الحدود العلوية (upper bounds) لأنواع معاملاتها، وسنتطرَّق إلى الأنواع المُعمَّمة بالتفصيل في الفصل العاشر، التجميعات.

## 15. الدوال النقيَّة

إن مفهوم «الدالة النقيّة» (pure functions) في البرمجة الوظيفيَّة هي الدالة التي تملك هاتين الصفتين:

- يجب أن تعيد الدالة المخرجات نفسها للمدخلات نفسها المعطاة إليها دومًا.
- لا يجب أن يكون للدالة أيَّة آثار جانبية.

تعني الصفة الأولى أنَّه عند استدعاء دالة عدة مرات مع تمرير القيم نفسها في كل مرة، يجب ألا تتغير القيمة التي تعيدها في كل استدعاء من تلك الاستدعاءات، ومثال ذلك هو الدالة abs،

فالقيمة المطلقة لعدد صحيح هي نفسها دائمًا من أجل ذلك العدد نفسه، فيمكن للدالة النقيَّة أن تعتمد فقط على المدخلات، ولكن لا تضطرها الحاجة إلى أن تستخدم جميع أنواع المدخلات.

أمَّا الصفة الثانية فتشير إلى أنَّه لا يجب على الدالة التسبب بأي تغييرات ملحوظة خارج نطاقها، لذلك لا يمكن أن تعتمد الدالة على أي حالة خارجيّة قابلة للتغيير أو تغيير قيمة متغيرات موجودة خارج نطاقها أو الكتابة على مجرى الدخل/الخرج (I/O).

إذا نُعِتت دالةٌ بالنقاوة، فيمكن تبديل الناتج الذي تعيده الدالة مكانها حيث استُدعيَت دون أن يتغير مجرى عمل

الشيفرة؛ فبالعودة مثلًا إلى الدالة التي تعيد القيمة المطلقة، أي شيفرة برمجيّة تعتمد على الاستدعاء `abs(-4)` يمكن أن نستبدل به القيمة 4 ولن يتغيَّر أي شيء في البرنامج.

انظر لكيفية استدعاء الدالة `impure` في المثال التالي:

```
val x = impure(5) + impure(5)
val y = 2 * impure(5)
```

قد تظن أنَّ قيمة `x` وقيمة `y` هي نفسها، والمتعارف عليه أنَّ `x + x` هي `x * 2` نفسها ولكن إذا نظرت إلى جسم الدالة `impure` الغير نقيَّة، فستجد ما لا يسرُّك:

```
val counter = AtomicInteger(1)
fun impure(k: Int): Int {
     return counter.incrementAndGet() + k
}
```

ستجد هنا أنَّ الدالة تغيِّر متغيِّر عام (global state)، أي حالة عامة)، لذلك فإن كل استدعاء يختلف عن الآخر، ويدل هذا المثال المقصود على الفرق بين الدوال النقيَّة والدوال غير النقيَّة.

تمتلك الدوال النقيَّة فوائد جمَّة، فيمكن تخزين نتائج الدوال في مخازن مؤقَّتة (caches)، وهذا مفيد للدوال البطيئة، ويمكن أيضًا استدعاء الدوال النقية بالتزامن مع بعضها بعضًا بسهولة، إذ لا يوجد أي شيء مشترك (حالة مشتركة) فيما بينها، ويمكن اختبارها بمعزل عن بعضها بعضًا بما أنَّها لا تعتمد على أي شيء سوى نُسِخ مدخلاتها (اختبار المثال السابق سيكون صعبًا مع آلية اختبار الوحدات [Unit test]).

سنغطي موضوع الدوال النقيَّة وكيف يمكن أن تساعد في عمليات الاختبار (testing) في الفصل العاشر، التجميعات.

## 16. جافا من وجهة نظر كوتلن

واحدة من أهم نقاط تفوق كوتلن على لغات JVM البديلة الأخرى (اللغات التي تُنتج تطبيقات تعمل على آلة جافا الافتراضية) هي أهميَّة التشغيل المتداخل بين كوتلن وجافا، فيمكن استدعاء معظم شيفرات جافا دون دعمٍ خاصّ مع كوتلن، ووصفنا بعض الحالات الخاصَّة في هذا الكتاب.

### أ. الجوالب والضوابط

هنالك عُرفٌ في جافا يدعى JavaBean ينص على أنَّ الحقول القابلة للتغيير تملك جالبًا (getter) وضابطًا (setter)، وتملك الحقول غير قابلة للتغيير على جالب فقط؛ فالجالب هو مجرد تابع لا يملك معاملات يبدأ اسمه بالكلمة get ثم تتبع باسم الحقل، وأما الضابط فهو تابع وحيد المعامل يبدأ اسمه بالكلمة set متبوعةً باسم الحقل، والمعامل المُمرَّر إليه هي القيمة التي يراد إعادة تعيينها للحقل:

```
public class Named {
     private String name;
     public String getName() {
            return name;
     }
     public void setName(String name) {
            this.name = name;
     }
}
```

هذا النمط قياسي في معظم جافا، ويمكن الوصول إلى التوابع المعرّفة بهذه الطريقة في كوتلن باستخدام صياغة نمط الخاصيّة (property-style syntax) الاعتيادي:

```
val named = Named()
println("My name is " + named.name)
named.name = "new name"
```

وبطبيعة الحال، يمكن الوصول إلى أسماء التابع بعدها دوالًا إذا رغبت بذلك.

| تنبيه | إذا احتوى حقل في جافا يحتوي على ضابط دون جالب، فلن تكون هذه الصياغة الخاصة متوفّرة. |
|---|---|

### ب. التوابع المجرَّدة الوحيدة

تعدُّ الواجهات التي تُعرّف تابعًا وحيدًا فقط أحد الأنماط الشائعة في جافا هي، ويمكنك أن ترى هذا في مكتبة

جافا القياسيّة، مثل Runnable و Callable و Closeable و Comparator ...إلخ. وغالبًا ما تُستخدَم في الأماكن التي يمكن فيها استخدام دالة وحيدة كانت تدعم دوال جافا في السابق، واعتمد الاسم «التابع المجرَّد الوحيد» (SAM، اختصار للعبارة Single Abstract Method) لوصف مثل هذه التوابع.

لمَّا كانت التوابع المجرّدة الوحيدة منتشرة في شتى الأمكنة، فتملك كوتلن دعم تحويل دالة مجهولة إلى تابع مجرَّد وحيد، وإذا كان التحويل لا لَبس فيه، فيمكنك ببساطة تمرير الدالة المجهولة حيث يتوقع وجود تابع مجرَّد وحيد:

```
val threadPool = Executors.newFixedThreadPool(4)
threadPool.submit {
    println("I don't have a lot of work to do")
}
```

في هذا المثال، يُعرَّف منفّذ مجمّع الخيوط (thread pool executor) لقبول نسخة النوع Runnable، ولذلك سيحوّل المصرّف الدالة المجهولة إلى نسخة Runnable لتمثّل تنفيذًا للتابع run، وستكون الشيفرة البرمجيّة المصرّفة مشابهة للمقتطف التالي:

```
threadPool.submit(object : Runnable {
    override fun run() {
        println("I don't have a lot of work to do")
    }
})
```

**تنبيه**

يعمل هذا الدعم الخاص فقط مع الواجهات وليس مع الأصناف المجرّدة (abstract classes) حتى لو ملك الصنف المجرّد تابعًا وحيدًا.

هل فكرت ماذا سيحدث إذا حَمَّل المستقبل تابعًا وحيدًا تحميلًا زائدًا؟ سيصعب آنذاك معرفة أي نوع من أنواع الدالة المجرَّدة الوحيدة يراد استعماله مكان الدالة المجهولة التي يراد تحويلها إليه، ويمكن ففي هذه الحالة تسهيل الأمر على المصرِّف عن طريق السماح له بمعرفة أي نوع تريده:

```
threadPool.submit(Runnable {
    println("I don't have a lot of work to do")
})
```

لاحظ أنَّ هذا المثال يشبه إلى حد كبير المثال الأول، باستثناء أننا ببساطة أرفقنا الدالة المجهولة باسم نوع التابع المجرَّد الوحيد (SAM type) الذي نرغب في استخدامه.

| تنبيه | لن تُنفِّذ كوتلن هذا التحويل على التوابع المجرَّدة الوحيدة (SAMs) المُعرَّفة في كوتلن نفسا، هذا لأنَّه يمكنك تعريف دالة تقبل دالة أخرى في كوتلن، مما يجعل هذا النوع من الأنماط لا حاجة إليه. |
|---|---|

### ت. تهريب معرِّفات كوتلن

بعض الكلمات المفتاحية في كوتلن هي معرّفات صالحة في جافا مثل `object` و `in` و `when` وغيرها، فإذا احتجت إلى استدعاء تابع من مكتبة جافا أو حقل باستخدام أحد الأسماء التالية، فلا يزال بإمكانك القيام بذلك من خلال تغليف الاسم مع الفاصلة العليا المائلة ( ` ` ).

فعلى سبيل المثال، لنفترض أن مكتبة جافا تُعرِّف الصنف والتابع التاليين:

```
public class Date {
    public void when(String str) { .... }
}
```

فإذا حدث هذا، فيمكنك استدعاؤه بالشكل التالي:

```
date.`when`("2016")
Checked exceptions
```

كما ناقشنا في وقت سابق، لا تملك كوتلن نوع الاستثناءات المُتحقَّق منها (Checked Exceptions)، ولذلك تُعامَل توابع جافا التي تملك استثناءات يُتحقَّق منها بنفس طريقة التوابع التي لا تفعل ذلك. لنفرض مثلًا أن التابع `createNewFile` مُعرَّف للتعامل مع الملفات:

```
public boolean createNewFile() throws IOException
```

يعني هذا في جافا أننا نحتاج إلى كتلة `try...catch...finally` أما في كوتلن فلا نفعل ذلك.

### ث. توابع void في جافا

نعرف الآن أنّ `void` في جافا تُشبه `Unit` في كوتلن، لذا يُعامَل أي تابع `void` في جافا على أنه دالة تُرجع `Unit`.

## 17. كوتلن من جافا

مثلما يمكن استخدام جافا بسلاسة في كوتلن، فيمكن استخدام كوتلن بسهولة في برامج جافا أيضًا.

### أ. الدوال الأعلى مرتبةً

لا تدعم آلة جافا الافتراضية (JVM) الدوال الأعلى مرتبةً (top-level functions)، لذلك يُنشئ مصرّف كوتلن صنف جافا مع اسم الحزمة لجعلها تعمل مع جافا وتُعرَّف الدوال آنذاك على أنها توابع جافا في هذا الصنف، والذي يجب إنشاء نسخة له قبل استخدامه. فعلى سبيل المثال، انظر إلى الدالة الأعلى مرتبةً التالية:

```
package com.packt.chapter4
fun cube(n: Int): Int = n * n * n
```

يولِّد مصرّف كوتلن في هذا المثال صنفًا يدعى `com.packt.chapter4.Chapter4` يحوي دوالًا توصف بأنها أعضاء ساكنة (static members). وإن أردنا استعمال هذه الدالة في جافا، يمكننا الوصول إليها بالطريقة التي نصل فيها إلى أي تابع ساكن آخر:

```
import com.packt.chapter4.Chapter4;
Chapter4.cube(3);
```

يمكننا حتى تغيير اسم الصنف الذي يُنشئه المصرّف باستخدام توصيف

صريح (annotation):

```
@file:JvmName("MathUtils")
package com.packt.chapter4
fun cube(n: Int): Int = n * n * n
```

يعمـــل المصـــرِّف في هـــذه الحالـــة على توليـــد تـــابع مقابـــل للدالـــة الأعلى مرتبـــةً في صـــنف يســـمى `com.pact.chapter4.MathUtils` بدلًا مما رأينا في الحالة السابقة، ويمكننا حتى فعل شيء أفضـل من هـذا بأن نخبر المصرّف بأننا نريد تجميع الدوال الأعلى مرتبةً من مختلف ملفات كوتلن في صنف جافا واحـد وذلـك عـبر التوصيف التالي:

```
@file:JvmMultifileClass
```

وهذا مفيد عندما تنتشر الدوال الأعلى مرتبةً في حزمة،و لكن تريد أن تستخدمها في جافا عـبر شـيفرة دخـول موحَّدة وبسيطة.

## ب. المعاملات الافتراضيّة

لا تدعم آلية جافا الافتراضـية (JVM) المعـاملات الافتراضـيَّة، لـذلك إن عُـرِّفت دالـة مـع معـاملات افتراضـية، يجب على المصرّف أن يُنشئ دالة وحيدة دون تلك المعاملات الافتراضـيّة؛ رغم ذلـك، يمكننـا أن نطلب من المصـرِّف إنشاء عدة بصمات لدالة تملك معاملات افتراضية تتناسب مع كل معامل افتراضي (أي زيادة تحميل الدالـة)، ويمكن لمستخدمي جافا بعد ذلك رؤية البصمات المختلفة لتلك الدالة واختيار الأنسب منها.

لنفترض أن لدينا التعريف التالي لدالة تملك معاملات افتراضية:

```
@JvmOverloads fun join(array: Array<String>, prefix: String =  "",
separator: String = "", suffix: String = "")
```

تجـــبر الواصـــفة `@JvmOverloads`، في هـــذا المثـــال، المصـــرِّف على توليـــد البصـــمات التاليـــة للدالة `join`:

```
public String join(String[] array) {
   return join(array, "");
}

public String join(String[] array, String prefix) {
   return join(array, prefix, "");
}

public String join(String[] array, String prefix, String separator) {
   return join(array, prefix, separator, "");
}

public String join(String[] array, String prefix, String separator, String
suffix) {
   //actual implementation
}
```

تُحمَّل الدالة تحميلًا زائدًا بهذا الشكل وتُنشَأ هذه البصمات في حال امتلكت معاملات افتراضيّة. تعمل هذه الخدعة الذكيّة مع البانيات والتوابع الساكنة.

### ت. الكائنات والتوابع الساكنة

تولّد الكائنات المسماة (named objects) والكائنات المرافقة (companion objects) في آلة جافا الافتراضية على أنها نُسَخ منفردة (singleton instances) لصنف معيَّن. على سبيل المثال، إذا عرَّفت كائنًا باسم Foo، فسينشئ مصرّف كوتلن صنفًا يسمى Foo يحتوي على حقل ساكن يدعى INSTANCE الذي سيحتوي على نسخة من Foo فقط:

```
object Console {
   fun clear() : Unit { }
}
```

يمكن استدعاء تابع كوتلن هذا من جافا كما يلي:

```
Console.INSTANCE.clear()
```

ومع ذلك، يمكنك أيضًا إبلاغ مصرّف كوتلن بأنك تريد تصريف هذه الدالة على أنَّها تابع ساكن في جافا بوسمها بالواصفة `@JvmStatic`:

```
object Console {
   @JvmStatic fun clear() : Unit { }
}
```

يمكننا الآن استدعاء هذا التابع من جافا باستخدام `INSTANCE` كما في السابق، إضافة إلى استدعائه مباشرةً من الصنف أيضًا:

```
Console.clear()
```

### ث. محو التسمية

لا تدعم آلة جافا الافتراضية الأنواع المُعمَّمة (generics) في شيفرة البايتكود، وهذا يعني أنه عندما تملك نوعًا مثل النوع list (قائمة)، أحد الأنواع المُعمَّمة، يمثَّل حينئذٍ النوعان `List<String>` و `List<Int>` كلاهما بنوع واحد أساسي أثناء عملية التصريف وستحدث في هذه الحالة كارثةٌ إن حوى هذان النوعان بصمات مختلفة لدالة واحدة.

إن فرضنا جدلًا وجود تعريفات (بصمات) الدالة التالية:

```
fun println(array: Array<String>): Unit {}
fun println(array: Array<Long>) : Unit {}
```

فسيُنتِج كلاهما البصمة نفسها.

كوتلن قادرة على التفريق بين هاتين البصمتين، ولكن جافا لا تستطيع ذلك. فإذا أردنا استخدامهما في جافا، فنستطيع فعل ذلك عبر إخبار المصرِّف بالاسم المراد استخدامه عند تصريف الشيفرة باستعمال الواصفة `@JvmName`:

```
@JvmName("printlnStrings")
fun println(array: Array<String>): Unit {}
@JvmName("printlnLongs")
fun println(array: Array<Long>) : Unit {}
```

يمكن الوصول إلى بصمات هذه الدالة في كوتلن وصولًا طبيعيًا باستعمال اسم الدالة `println` فقط، وإن أردنا استعمال مختلف بصماتها في جافا، فنستعمل الأسماء المحدَّدة في الواصفة، أي الاسم `printlnStrings` للبصمة الأولى والاسم `printlnLongs` للبصمة الثانية.

## ج. الاستثناءات المُتحقَّق منها

يمكننا في جافا التقاط (catch) الاستثناءات (الاعتراضات) المُتحقَّق منها (checked exceptions) إذا صُرِّح عنها في التابع، وحتى لو ألقى جسم التابع هذا الاستثناء، لذلك إذا كان لدينا دالة ستُستخدَم من جافا ونرغب في السماح للآخرين بالتقاط الاستثناءات التي تلقيها، يجب أن نبلِّغ المصرِّف بضرورة إضافة الاستثناء إلى بصمة التابع. يمكننا فعل هذا باستخدام الواصفة `@Throws`:

```
@Throws(IOException::class)
fun createDirectory(file: File) {
     if (file.exists())
            throw IOException("Directory already exists")
     file.createNewFile()
}
```

في هذا المثال، يمكن استخدام الدالة `createDirectory` الآن في كتلة `try...catch...finally` من جافا:

```
try {
     Chapter4.createDirectory(new File("mobydick.txt"));
} catch (IOException e) {
     // handle exception here
}
```

## 18. خلاصة الفصل

الدوال هي اللبنة الأساسيَّة لأية لغة حديثة، وجاءت أغلب مزايا كوتلن مُعززةً مبدأ «خير الكلام ما قل ودل» ومبدأ «الإيجاز خلاصة الذكاء» متيحةً للمطورين إنجاز ما يريدون بأقصر شيفرة ممكنة. وتعدُّ الدوال مفتاحًا لكتابة تعابير قويَّة خصوصًا الأعلى مرتبةً منها كما سنرى في الفصل القادم.

الفصل الخامس:

# الدوال الأعلى مرتبةً والبرمجة الوظيفيَّة



تحدثّنا في الفصل السابق عن دعم كوتلن للدوال والمميزات المختلفة التي يمكننا استخدامها عند كتابة الدوال، وسنستمر في هذا الفصل بالخوض في هذا الموضوع عن طريق مناقشة الدوال الأعلى مرتبةً (higher order functions) وكيف يمكننا استخدامها لكتابة شيفرات برمجيّة أنظف وأكثر تعبيرًا.

سنغطي في هذا الفصل:

- الدوال الأعلى مرتبةً (Higher order functions) والمغلفات (closures)
- الدوال المجهولة (Anonymous functions)
- مراجع الدوال (Function references)
- تعبيرات البرمجة الوظيفيّة (Functional programming idioms)
- تخصيص اللغات مخصصة المجال (Custom DSLs)

## 1. الدوال الأعلى مرتبةً

الدالة الأعلى مرتبةً (higher order function) هي ببساطة دالة إمّا أن يكون أحد معاملاتها دالةً أخرى، أو ترجع دالة بعدّها قيمةً معادةً أو كليهما. إليك المثال التالي:

```
fun foo(str: String, fn: (String) -> String): Unit {
      val applied = fn(str)
      println(applied)
}
```

عرّفنا هنا دالة باسم foo مع معاملين، الأول هو سلسلة نصيّة، والثاني هو دالة من سلسلة نصيّة إلى سلسلة نصيّة، ونقصد بهذا أن مدخلات الدالة تقبل سلسلة نصيّة ومخرجاتها سلسلة نصيّة أخرى؛ ولاحظ أيضًا الصياغة المستخدمة في تعريف معامل الدالة، إذ غُلِّفَت نوع الدخل بين قوسين وفُصِل نوع الخرج بسهم صغير.

إن أردنا استدعاء هذه الدالة، يمكننا تمرير دالة مجهولة إليها (راجع الدوال المجهولة التي تحدثنا عنها في الفصل الرابع إن نسيتها):

```
foo("hello", { it.reversed() })
```

كما ترى، السلسلة التي مرّرنها هي hello، ولقد أعدنا تمريرها إلى الدالة المجهولة التاليّة، والتي تعيدها معكوسةً، وستكون نتيجة هذا الاستدعاء olleh، وتذكر أنه يمكن استعمال it مع الدالة المجهولة التي تملك معاملًا وحيدًا لتجنّب تسمية المعامل صراحةً وللاختصار.

قد تتساءل في هذه المرحلة لماذا قد نستفيد من ذلك؟ فبعد كل شيء، يمكننا كتابة شيفرة برمجيّة كما يلي:

```
fun foo2(str: String) {
     val reversed = str.reversed()
     println(reversed)
}
```

نتيجة الشيفرة البرمجية السابقة نفسها، ومع ذلك، فإن مزايا الدوال ذات المرتبة الأعلى واضحةٌ عندما نريد كتابة دالة يمكن أن تعمل في حالات مختلفة، لنفترض أننا نرغب في ترشيح العناصر من قائمة list إلى عناصر فرديّة وزوجيّة، فستكون الشيفرة البرمجيّة مشابهة لهذه:

```
val ints = listOf(1, 2, 3, 4, 5, 6)
val evens = mutableListOf<Int>()
val odds = mutableListOf<Int>()
for (k in ints) {
     if (k % 2 == 0)
            evens.add(k)
     else
            odds.add(k)
}
```

تُضاف كل قيمة إلى قائمة أخرى عند التكرار، وسنُطبّق عامل باقي القسمة modulo لفصل الأعداد الفردية عن الزوجية.

ومع ذلك، يمكننا استخدام الدوال الأعلى مرتبةً بدلًا من ذلك، كما يلي:

```
val ints = listOf(1, 2, 3, 4, 5, 6)
val odds = ints.filter { it % 2 == 1 }
val evens = ints.filter { it % 2 == 0 }
```

يملك هذا النوع من الشيفرات البرمجيّة الخاصيّة النادرة في أن يكون سريع الكتابة وسهل القراءة، ولديه فائدة أن نتائج الدالتين evens و odds غير قابلة للتغيير كما لو أنها ضمن اللغة.

| تنبيه | سنشرح التجميعات والدوال الأعلى مرتبةً المرتبط بها شرحًا كاملًا في الفصل العاشر: التجميعات. |
| --- | --- |

## أ. إعادة دالة

لنعد الآن إلى تعريف الدالة ذات المرتبة الأعلى (Higher Order Function)، تذكّر أننا قلنا تعدُّ الدالة التي تُعيد دالةً أخرى دالةً عالية المرتبة دون أي خلاف:

```
fun bar(): (String) -> String = { str -> str.reversed() }
```

رأينا في المثال أنَّ استدعاء bar() سيُعيد دالةً تقبل سلسلة نصيّة وتعيد السلسلة النصيّة معكوسةً.

ولإعادة دالة، سنستخدم علامة مساواة بعد النوع المعاد، وسنُغلِّف الدالة بين قوسين معقوصين؛ ومن الناحيّة التقنيَّة، هذه دالة سطرية (مكتوبة في سطر واحد)، إذ أنَّ التعبير الموجود بعد علامة المساواة هو جسم الدالة.

يمكننا استدعاء هذه الدالة على النحو التالي:

```
val reversi = bar()
reversi("hello")
reversi("world")
```

أسندنا في هذا المثال الدالة bar إلى متغير يدعى reversi قبل استدعائه مع قيمتين مختلفتين.

فائدة هذا الأسلوب نلتمسها عندما يكون لدينا دالة تقبل عدة قيم ثم تعيد دالةً تستخدم المدخلات التي مُرِّرت إلى الدالة الأصليّة؛ لنرجع إلى مثال الترشيح، ونعرّف دالةً ترشح الأعداد المُمرَّرة إليها بناءً على باقي القسمة:

```
fun modulo(k: Int): (Int) -> Boolean = { it % k == 0 }
```

لاحظ أن القيمة k المدخلة تُستخدم في الدالة المعادة، ويمكن الآن دمجها مع الدالة filter عالية المرتبة

الموجودة في صنف القائمة (list):

```
val ints = listOf(1, 2, 3, 4, 5, 6)
val odd = ints.filter(modulo(1))
val evens = ints.filter(modulo(2))
val mod3 = ints.filter(modulo(3))
```

**تنبيه**

لم نستخدم هنا القوسين المعقوصين كما في المثال السابق؛ فإذا فعلنا ذلك، فستُعرَّف آنذاك دالةٌ أخرى تستدعي الدالة `modulo`، مما يعطينا دالة من دالة.

### ب. إسناد دالة

يمكن إسناد الدوال أيضًا إلى متغيرات لتسهيل عملية تمريرها إلى دوال أخرى:

```
val isEven: (Int) -> Boolean = modulo(2)
listOf(1, 2, 3, 4).filter(isEven)
listOf(5, 6, 7, 8).filter(isEven)
```

استخدمنا الدالة `modulo` التي عرّفناها سابقًا في هذا المثال، إذ أسندنا نسخةً منها إلى المتغيّر `isEven`؛ صُرِّح عن النوع المعاد بوضوح ويمكن للقارئ معرفته بسهولة، ولكن عادةً ما يُحذف، وتُستخدَم نسخة من هذه الدالة آنذاك مرّتين.

يمكننا أيضًا إسناد الدوال المجهولة إلى متغيّرات، وسنحتاج حينئذٍ إلى مساعدة المصرِّف مع أنواع المعاملات إما بالشكل التالي:

```
val isEven : (Int) -> Boolean = { it % 2 == 0 }
```

أو بالطريقة التالية:

```
val isEven = { k : Int -> k % 2 == 0 }
```

قد يعدُّ ذلك مُفيدًا إذا احتجت إلى إعداد دالة ذات شأن أو تستهلك وقتًا طويلًا ويراد استخدامها عدة مرات.

يقال أن اللغات التي تدعم الدوال الأعلى مرتبةً وتدعم عملية إسناد الدوال تدعم الدوال الكيانيَّة (first class

functions)[8].

## 2. المُغلِّفات

يُعدُّ المُغلِّف (closure) في البرمجة الوظيفيَّة دالةً تملك الوصول إلى متغيرات ومعاملات معرَّفة في نطاقات خارجيّة أكبر، إذ يقال أن الدالة تقترب (close over) من هذه المتغيرات وتغطيها/تغلِّفها، وبالتالي يطلق عليها التسمية «مُغلِّف».

لنفترض أننا نرغب في تحميل أسماء من قاعدة البيانات وترشيحها لاستخلاص تلك التي تطابق بعض معايير البحث فقط، لذا سنستخدم صديقنا القديم التابع `filter`:

```
class Student(val firstName: String, val lastName: String)
fun loadStudents(): List = ...
    // تحميل من قاعدة البيانات
fun students(nameToMatch: String): List<Student> {
    return loadStudents().filter {
        it.lastName == nameToMatch
    }
}
```

لاحظ أنَّ الدالة المجهولة المُمرَّرة إلى التابع `filter` تستخدم المعامل المعطى للدالة الخارجيّة، وهذا المعامل مُعرَّف في نطاق خارج نطاقها، وبالتالي تسعى هذه الدالة للاقتراب من ذلك المعامل وتغليفه ضمن نطاقها.

يمكن للمُغلِّف الوصول إلى المتغيرات المحليّة أيضًا:

```
val counter = AtomicInteger(0)
val cores = Runtime.getRuntime().availableProcessors()
val threadPool = Executors.newFixedThreadPool(cores)
threadPool.submit {
    println("I am task number ${counter.incrementAndGet()}")
}
```

8 يقال إن اللغة تدعم الدوال الكيانية عندما تُعامَل الدوال في تلك اللغة مثلها كمثل أي متغيِّر، أي يمكن عدَّها معاملات لتُمرَّر إلى دوال أخرى ويمكن أن تعيدها دوال أخرى ويمكن أن تُسنَد إلى متغيّر.

نرسل في هذا المثال عددًا من المهام إلى مجمَّع خيوط (thread pool)، إذ تملك كل مهمة، كما ترى، إمكانيّة الوصول إلى عداد مشترك (باستخدام AtomicInteger لسلامة الخيط [thread]).

يمكن أن تؤدي المُغلّفات أيضًا إلى التلاعب بقيم المتغيرات التي تغلفها وإفسادها:

```
var containsNegative = false
val ints = listOf(0, 1, 2, 3, 4, 5)
ints.forEach {
     if (it < 0)
            containsNegative = true
}
```

ببساطة، تقترب هذه الشيفرة البرمجيّة من المتغيّر المحلي containsNegative، وتغيِّر قيمته إلى true إذا وُجدت قيمة سالبة في القائمة. يمكنك عمليًا استخدام دالة مبنية مسبقًا (built-in function) لهذا الغرض بدلًا من هذه الدالة، لكن المثال يوضِّح كيفيّة تحديث قيم المتغيرات من داخل دالة مجهولة.

## 3. الدوال مجهولة الاسم

نستدعي في كثير من الأحيان الدوال عالية المرتبة باستخدام دوال مجرَّدة (function literals)، خاصةً إذا كانت الدالة قصيرة:

```
listOf(1, 2, 3).filter { it > 1 }
```

كما ترى، لا يوجد سبب لتعريف الدالة المُمرَّرة إلى الدالة filter في أي مكان آخر؛ وعند استخدام الدوال المُجرَّدة بهذا الشكل، فلن نتمكن من ضبط نوع القيمة المعادة، وهذا في العادة ليس بمشكلة إذ سيستنتج مصرّف كوتلن نوع القيمة المعادة.

على أية حال، قد نرغب في بعض الأحيان أن نكون صريحين وواضحين بشأن نوع القيمة المعادة، فيمكننا آنذاك استخدام ما يسمى بالدالة مجهولة الاسم (anonymous function)[9]، إذ تشبه هذه الدالة تعريف الدالة

9 لا تخلط بينها وبين الدالة المجرَّدة (function literal)، التي يطلق عليها أحيانًا «دالة مجهولة» أيضًا.

العاديّة، باستثناء حذف الاسم فقط:

```
fun(a: String, b: String): String = a + b
```

يمكن أن يُستخدَم الدالة مجهولة الاسم بالطريقة التالية:

```
val ints = listOf(1, 2, 3)
val evens = ints.filter(fun(k: Int) = k % 2 == 0)
```

إذا كان من الممكن استنتاج نوع المعامل، فيمكن حذف النوع ليُكتَب المثال السابق بالشكل التالي:

```
val evens = ints.filter(fun(k) = k % 2 == 0)
```

## 4. مراجع الدالة

رأينا حتى الآن في هذا الفصل كيف يمكن تمرير الدوال بعدّها معاملات، والطرائق التي استخدمناها لفعل ذلك هي إما عن طريق إنشاء دوال مجرَّدة (function literals)، أو باستخدام دالة تُرجع دالة أخرى.

### أ. مراجع الدوال الأعلى مرتبةً

لكن ماذا لو كان لدينا دالة عالية المرتبة (top-level function) ونريد استخدامها؟ يمكننا تغليف الدالة في دالة أخرى بالطبع:

```
fun isEven(k: Int): Boolean = k % 2 == 0
val ints = listOf(1, 2, 3, 4, 5)
ints.filter { isEven(it) }
```

ويمكننا بدلا من ذلك استخدام ما يسمى مرجع الدالة (function reference)، فباستخدام نفس تعريف الدالة `isEven` السابق، يمكننا كتابته على النحو التالي:

```
val ints = listOf(1, 2, 3, 4, 5)
ints.filter(::isEven)
```

لاحظ أن صياغة `::` تُستخدم قبل اسم الدالة.

### ب. مراجع الدوال التابعة والمُوسّعة

يمكن استخدام مراجع الدوال التابعة (Member function) والدوال المُوسّعة (extension function) عن طريق وضعهم كبادئة مع اسم الصنف، فدعنا نعرّف دالة مُوسّعة تتعامل مع الأعداد الصحيحة تسمى `isOdd`، كما في المثال التالي:

```
fun Int.isOdd(): Boolean = this % 2 == 0
```

يمكننا استخدام هذا داخل دالة مجرَّدة مثل المعتاد:

```
val ints = listOf(1, 2, 3, 4, 5)
ints.filter { it.isOdd() }
```

يمكننا أيضًا استخدام مرجع إليها بدلًا من ذلك:

```
val ints = listOf(1, 2, 3, 4, 5)
ints.filter(Int::isOdd)
```

| تنبيه | يملك مرجع دالة إلى دالة تابعة أو دالة مُوسّعة معامل إضافي، وهو نسخة من أو المستقبل نفسه الذي استدعي معها. |
|---|---|

قد تبدو مراجع الدالة المُجرَّد طريقة أخرى للقيام بنفس الشيء، لكن ضع في اعتبارك حالة الدالة التي تقبل معاملات مُتعدِّدة:

```
fun foo(a: Double, b: Double, f: (Double, Double) -> Double) =  f(a, b)
```

هنا، ستستدعى الدالة `foo` معامل الدالة مع المدخلات `a` و `b`، ولاستدعاء هذا، يمكننا بالطبع، تمرير دالة مجرَّدة:

```
foo(1.0, 2.0, { a, b -> Math.pow(a, b) })
```

الدالة `Math.pow` هي دالة تابعة، وبما أننا نعرف أنها تقبل معاملين اثنين من النوع `double` وتُرجع عددًا آخر

من النوع double، فيمكننا استخدام مرجع دالة، وسيكون لهذا بصمة دالة مطابقة، وبهذا يقلّل من كتابة الشيفرات البرمجيّة:

```
foo(1.0, 2.0, Math::pow)
```

### ت. مراجع الربط

في كوتلن 1.1، يمكننا الحصول على مراجع دالة مرتبطة إلى نسخة معيَّنة، وهذا يعني أنَّه يمكننا وضع تعبير قبل عامل ::، ومن ثم سيرتبط المرجع إلى تلك النسخة المعيّنة، وهذا يعني، بخلاف المراجع غير المرتبطة (unbound references)، ولن يزيد رُتبيّة (arity) الدالة المعادة.

وازن بين المثالين التاليين، إذ يستخدم الأول مراجع غير مضمَّنة:

```
fun String.equalsIgnoreCase(other: String) = this.toLowerCase() ==
other.toLowerCase()
listOf("foo", "moo", "boo").filter {
     (String::equalsIgnoreCase)("bar", it)
}
```

لدينا دالة بسيطة لحالة المساواة غير الحساسة، لكن عند إنشاء مرجع دالة إليها، فهي تملك البصمة (String, String) -> Boolean، فالمعامل الأول هو المتلقي، وهذا يعني أنه لا يمكننا ببساطة تمرير المرجع إلى دالة مُرشِّحة في القائمة، ولكن بدلًا من ذلك تغلفها مرَّة أخرى في دالة مُجرَّدة أخرى.

دعنا نحاول مرَّة أخرى، وهذه المرة باستخدام مرجع الربط:

```
fun String.equalsIgnoreCase(other: String) = this.toLowerCase() ==
other.toLowerCase()
listOf("foo", "baz", "BAR").filter("bar"::equalsIgnoreCase)
```

يمكننا باستخدام نفس التعريف للدالة equalsIgnoreCase إنشاء مرجع ربط على المتلقي، الذي هو bar، وينتج لنا هذا مع البصمة (String) -> Boolean، ويملك هذا المرجع الشكل الصحيح لتمريره مباشرةً إلى الدالة filter.

## 5. مستقبلات الدالة المُجرَّدة

تعلمنا في الفصل السابق أن مستقبل أي دالة هو النسخة التي تشير إليها الكلمة المفتاحية this عندما تكون داخل جسم الدالة. ففي كوتلن، يمكن تعريف معاملات الدالة لقبول متلقي عند استدعائها، ونفعل ذلك باستخدام الصياغة التالية:

```
fun foo(fn: String.() -> Boolean): Unit
```

وبعد ذلك، عند استدعاء الدالة fn في جسم الدالة foo، يُطلب منا استدعاؤها على نسخة من سلسلة نصيَّة، كما يمكنك أن ترى إذا أكملنا تعريف الدالة foo:

```
fun foo(fn: String.() -> Boolean): Unit {
     "string".fn()
}
```

تعمل هذه الميِّزة أيضًا مع الدوال المجهولة:

```
val substring = fun String.(substr: String): Boolean =
this.contains(substr)
"hello".substring("ello")
```

قد تفضِّل صياغة الدالة المجهولة إذا كنت ترغب في إسناد دالة إلى متغيِّر، كما في السابق، وهذا لأن المتلقي لا يمكن تحديده باستخدام دالة مُجرَّدة.

تعد مستقبلات الدوال مفيدة عند كتابة DSL مخصَّصة، وسنغطي هذا بالتفصيل في وقتٍ لاحق.

## 6. الدوال في آلة جافا الافتراضية JVM

قبل الإصدار 8 من آلة جافا الافتراضيّة، لم تكن تُدعم الدوال الكيانيَّة (First class functions)، وبما أن كوتلن تستهدف جافا 6 للتوافق مع أجهزة أندرويد، فكيف يعالج المُصرِّف الدوال؟

تبيَّن أن جميع الدوال في كوتلن تُصرَّف إلى نسخ من أصناف تسمى Function0، و Function1، و Function2 وهكذا إذ يمثِّل الرقم في اسم الصنف عدد المدخلات، وإذا نظرت إلى النوع داخل البيئة المتكاملة IDE، فستتمكّن من معرفة أي صنف تُصرَّف إليه الدالة؛ فعلى سبيل المثال، الدالة ذات البصمة (Int)→Boolean

سيظهِر النوع على شكل `Function1<Int, Boolean>`، حيث يملك كل واحد من أصناف الدالة استدعاء دالة تابعة تُستخدَم لتعريف جسم الدالة.

إليك تعريف `Function0` من الشيفرة المصدريّة لكوتلن، والتي لا تقبل معاملات إدخال:

```
/** A function that takes 0 arguments. */
public interface Function0<out R> : Function<R> {
     /** Invokes the function. */
     public operator fun invoke(): R
}
```

هذا تعريف `Function1` والذي يقبل معامل إدخال واحد:

```
/** A function that takes 1 argument. */
public interface Function1<in P1, out R> : Function<R> {
     /** Invokes the function with the specified argument. */
     public operator fun invoke(p1: P1): R
}
```

تملك جميع النسخ نوعًا معادًا، وهو نوع المعامل المتوضع في أقصى اليمين، وستتبع بقية تعريفات `FunctionN` هذا المنطق.

## أ. بايتكود

وكمثال على الناتج الذي سيصدره مصرِّف كوتلن، يمكننا عرض البايتكود المولَّد لاستدعاء دالة بسيطة، ودعنا نستخدم دوال مُرشِّحة لعدد صحيح الذي استخدمناه من قبل.

انظر إلى هذه الدالة البسيطة:

```
val isEven: (Int) -> Boolean = { it % 2 == 0 }
```

ستنتج هذه الدالة البسيطة بايتكود التالي:

```
final class com.packt.chapter5._5_x_inlineKt$test$isEven$1 extends
kotlin.jvm.internal.Lambda implements
kotlin.jvm.functions.Function1<java.lang.Integer, java.lang.Boolean> {
```

```
public static final com.packt.chapter5._5_x_inlineKt$test$isEven$1
INSTANCE;
public java.lang.Object invoke(java.lang.Object);
Code:
   0: aload_0
   1: aload_1
   2: checkcast     #11      // class java/lang/Number
   5: invokevirtual #15       // Method  java/lang/Number.intValue:()I
   8: invokevirtual #18       // Method invoke:(I)Z
   11: invokestatic  #24      // Method  java/lang/Boolean.valueOf:
(Z)Ljava/lang/Boolean;
   14: areturn
public final boolean invoke(int);
 Code:
0: iload_1
1: iconst_2
2: irem
3: ifne          10
6: iconst_1
7: goto          11
10: iconst_0
11: ireturn
com.packt.chapter5._5_x_inlineKt$test$isEven$1();
Code:
0: aload_0
1: iconst_1
2: invokespecial #33        // Method
kotlin/jvm/internal/Lambda."<init>":(I)V
5: return
static {};
 Code:
   0: new           #2   // class
com/packt/chapter5/_5_x_inlineKt$test$isEven$1
3: dup
4: invokespecial #53    // Method "<init>":()V
```

```
7: putstatic      #55    // Field
INSTANCE:Lcom/packt/chapter5/_5_x_inlineKt$test$isEven$1;
10: return
}
```

يمكنك أن ترى في السطر الأول أن هذا الصنف موسع:

```
kotlin.jvm.functions.Function1<java.lang.Integer, java.lang.Boolean>
```

يتطابق هذا مع نوع الدالة الذي عرَّفناها، وستلاحظ أيضًا أن هنالك استدعاء دوال تحتوي على منطق الدالة، وتسمى `iconst_2` و `irem` والتي تنفِّذ العملية `modulo(2)`.

بقيّة البايتكود معني بالسماح لاستدعاء الدالة كتابع ثابت، وبما أن الدوال لا تحتوي على حالة بخلاف مدخلاتها، فيمكن تصميمها على أنها نموذج المفردة عن طريق تابع ثابت.

| تنبيه | تُنفّذ عمليات المُغلّفات (Closures) عن طريق زيادة رتبيّة (arity) الدالة لقبول معاملات إضافيّة، والتي هي متغيّرات المُغلّف، ويدرج المصرِّف هذا تلقائيًا. |
|---|---|

## 7. دالة مركبة

لقد رأينا كيف يمكننا استخراج قيمة دالة من مستوى أعلى موجود أو دالة مُوسّعة، وستكون الخطوة التالية هي الوظيفة التي تسمح لنا بالجمع بين عدة دوال معًا بطريقة موجزة وهذه الدالة تدعى بدالة مركبّة (function composition).

على عكس العديد من اللغات الأخرى، لا تملك كوتلن أي دعم مدمج للدوال المركبة، ومع ذلك، من السهل إضافة واحدة خاصة بنا باستخدام الوسائل التي رأينها لحد الآن للتلاعب في الدوال.

يمكننا البدء بتعريف دالة مركبة تقبل دالتي إدخال، وسترجع دالة جديدة تستدعيهما بالتوالي عند التطبيق، وبالطبع، يجب أن تكون نوع مخرجات الدالة الأولى مطابقة لنوع مدخلات الدالة الثانية:

```
fun <A, B, C> compose(fn1: (A) -> B, fn2: (B) -> C): (A) -> C = { a ->
     val b = fn1(a)
```

```
        val c = fn2(b)
        c
    }
```

لقد كتبت هذا المثال بطريقة مطوّلة إلى حد ما، وذلك بتعيين كل خطوة مع الدالة الخاصة بها، لكن يوضح هـذا للقارئ ما يجري، الدالة التي سترجع هي تـركيب من A إلى C، والـتي حقّقناهـا من خلال اسـتدعاء A إلى B ومن ثم B إلى C.

يمكننا استدعاء هذه الدالة بسهولة على النحو التالي:

```
val f = String::length
val g = Any::hashCode
val fog = compose(f, g)
```

هنا، اشتققنا مراجع دالتيّن، الأولى للحصـول على طـول السلسـلة النصـية، والثانيـة للحصـول على قيمـة hash لهذا الطول، وبمجرد دمجهما معًا، يمكننا استدعاء fog عن طريق تطبيق سلسلة نصية:

```
fog("what is the hash of my length?")
```

**تنبيه**

هذا ليس دالة مركبة بالمعنى الرياضي، والذي يطبّق الدالة على اليمين أولًا ومن ثم الدالة على اليسار للوصول إلى النتيجة، وبدلًا من ذلك، نطبق الدوال في ترتيب من اليسار إلى اليمين.

تذكر أن جميع الدوال تصرّف إلى نسخ لأصناف FunctionN مدمجة، ويمكننا الاستفادة من هذا لإنشـاء دوال مُوسّعة على هذه الأصناف، دعنـا نعيـد صيـاغة تـركيب الدالـة لاسـتخدام صيـاغة infix مـع عامـل مناسـب لجعل التركيب أسهل للاستخدام.

بما أننا نعلم أنه لا يمكن تعريف دوال infix إلا كدوال أعضاء أو دوال مُوسّعة، ونحتـاج إلى تغيـير compose لتعريفها على أنها دالة مُوسّعة على نسخة FunctionN المناسبة:

```
infix fun <P1, R, R2> Function1<P1, R>.compose(fn: (R) -> R2): R2 = {
```

```
    fn(this(it))
}
```

يسمح لنا هذا باستدعاء شيفرة برمجيّة مشابهة لما يلي:

```
val f = String::length
val g = Any::hashCode
val fog = f compose g
```

لنحدّث الآن هذا لاستخدام عامل، ففي هذه المرحلة، لا يوجد شيء أكثر تعقيدًا من استبدال الاسم مع اسم عامل التعيين، وإضافة الكلمة المفتاحية `operator`:

```
operator infix fun <P1, R, R2> Function1<P1, R>.times(fn: (R) → (P1) -> R2
= {
    fn(this(it))
}
```

يمكنك استدعاؤها كالتالي:

```
val f = String::length
val g = Any::hashCode
val fog = f * g
```

لا يختلف هذا كثيرًا عن اللغات التي تحتوي على دعم مضمَّن للدوال المركبة.

## 8. الدوال المباشرة

كما رأينا في الأقسام السابقة، فإن الدوال هي أمثلة كائنات وبالطبع، يتطلّب كل نسخة تخصيص مكان في الكومة (heap) ويتطلّب أيضا استدعاءات تابع عند استدعاء الدالة، وعمومًا، استخدام الدوال يؤثر على الأداء.

تسمح لنا كوتلن بتجنّب هذا العبء باستخدام الكلمة المفتاحية `inline`، فهذه الكلمة تخبر المصرّف أن الدالة التي عليها علامة `inline`، بالإضافة إلى معاملاتها أيضا، ينبغي توسيعها وتوليدها بشكل مباشر وآني وقت الاستدعاء، ومن هنا جاءت التسمية، دالة مباشرة (Inline function).

ماذا يعني هذا بالضبط؟ لنفترض أن لدينا دالة تتعامل مع الموارد بطريقة آمنة: وبهذا سيُغلَق المورد دائمًا بشكل

صحيح، وحتى لو كانت الشيفرة البرمجيَّة ترمي استثناء:

```
fun <T : AutoCloseable, U> withResource(resource: T, fn: (T) -> U): U {
     try {
            return fn(resource)
     } finally {
            resource.close()
     }
}
```

كما ترى، لقد غلفنا تطبيق معامل الدالة في كتلة `try...finally` فقط، فهذه الدالـة بسـيطة للغايـة، وتعمـل ببساطة على إزاحة بعض الشيفرات البرمجيّة المتكـرِّرة كلما أردنـا اسـتخدام مـورد قابـل للإغلاق، يمكننـا اسـتخدام التالي:

```
fun characterCount(filename: String): Int {
     val input = Files.newInputStream(Paths.get(filename))
     return withResource(input) {
            input.buffered().reader().readText().length
     }
}
```

تفتح هذه الدالة ملفًا، وتقرأ النص وتحسـب عـدد الأحـرف وتسـتخدم `withResource` لضـمان إغلاق مجـرى الإدخال بشكل صحيح إذا رُمي استثناء.

عند تصريف هذه الشيفرة البرمجيّة، ينتهي بنا الأمر إلى إنشاء معامل دالـة كنسـخة، إذا نُفّذت شـيفرة برمجيّـة مثل هذه عدة مرات في الحلقة، فستضاف هذه الأجزاء – المعاملات- إلى الذاكرة وتحجز مكانًا لها.

لنقِ نظرة على البايتكود الباني بواسطة المصرِّف للدالة `characterCount`:

```
0: aload_0
1: ldc            #37      // String filename
3: invokestatic   #15               // Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/
Object;Ljava/lang/String;)V
6: aload_0
```

```
7: iconst_0
8: anewarray     #39      // class java/lang/String
11: invokestatic  #45                // Method java/nio/file/Paths.get:
(Ljava/lang/String;[Ljava/lang/String;)Ljava/nio/file/Path;
14: iconst_0
15: anewarray     #47                // class java/nio/file/OpenOption
18: invokestatic  #53                // Method
java/nio/file/Files.newInputStream:(Ljava/nio/file/Path;[Ljava/nio/file/
OpenOption;)Ljava/io/InputStream;
21: astore_1
22: aload_1
23: checkcast     #25                // class java/lang/AutoCloseable
26: new           #55                // class
com/packt/chapter5/_5_x_inlineKt$first$1
29: dup
30: aload_1
31: invokespecial #59                // Method
com/packt/chapter5/_5_x_inlineKt$first$1."<init>":(Ljava/io/InputStream;)V
34: checkcast     #19                // class
kotlin/jvm/functions/Function1
37: invokestatic  #61                // Method
withResource:(Ljava/lang/AutoCloseable;Lkotlin/jvm/functions/Function1;)Lj
ava/lang/Object;
40: checkcast     #63                // class java/lang/Number
43: invokevirtual #67                // Method java/lang/Number.intValue:
()I
46: ireturn
```

بالنسبة لغير المعتاد على مخرجات البايتكود، فإن الجزء الذي يهمنا هو في السطر 26، ويمكنك أن ترى أن نسخة جديد لدالة مُجرَّدة أُنشِئت قبل تمريرها إلى الدالة `withResource` في السطر 37، والصنف الذي يحتوي على الشيفرة المصدريَّة للدالة المُجرَّدة يدعى `com/packt/chapter5/_5_x_inlineKt$first$1`.

إذا وضعنا علامة `inline` على الدالة `withResource` لنشير إلى أنها مباشرة، فإن مصرِّف كوتلن لن يولّد هذه كاستدعاء نسخة جديد، وبدلًا من ذلك سيوّلد الشيفرة البرمجية في موقع الاستدعاء.

أولًا، نضع توصيف للدالة باستخدام الكلمة المفتاحية:

```
inline fun <T : AutoCloseable, U> withResource(resource: T, fn:  (T) ->
U): U {
   try {
      return fn(resource)
   } finally {
      resource.close()
   }
}
```

سيترجم المصرِّف استدعاء `characterCount` إلى التالي:

```
fun characterCountExpanded(filename: String): Int {

   val input = Files.newInputStream(Paths.get(filename))

      return withResource(input) {
input.buffered().reader().readText().length

   }
}
```

هذا هو هدفنا قبل التعرف على الدوال المساعدة، والآن مخرجات البايتكود لـ `characterCount` قـد تغيَّرت بشكل ملحوظ:

```
0: aload_0
1: ldc            #48            // String filename
3: invokestatic   #15            // Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/
Object;Ljava/lang/String;)V
6: aload_0
7: iconst_0
8: anewarray      #50            // class java/lang/String
11: invokestatic   #56           // Method
java/nio/file/Paths.get:(Ljava/lang/String;[Ljava/lang/String;)Ljava/nio/
file/Path;
```

```
14: iconst_0
15: anewarray     #58          // class java/nio/file/OpenOption
18: invokestatic  #64          // Method
java/nio/file/Files.newInputStream:(Ljava/nio/file/Path;[Ljava/nio/file/
OpenOption;)Ljava/io/InputStream;
21: astore_1
22: nop
23: nop
24: aload_1
25: checkcast     #31          // class java/lang/AutoCloseable
28: checkcast     #66          // class java/io/InputStream
31: astore_2
32: aload_1
33: invokevirtual #70          // Method java/io/InputStream.read:()I
36: istore        4
38: aload_1
39: checkcast     #31          // class java/lang/AutoCloseable
42: invokeinterface #35,  1       // InterfaceMethod
java/lang/AutoCloseable.close:()V
47: iload         4
49: goto          66
52: astore        4
54: aload_1
55: checkcast     #31                 // class java/lang/AutoCloseable
58: invokeinterface #35,  1      // InterfaceMethod
java/lang/AutoCloseable.close:()V
63: aload         4
65: athrow
66: ireturn
```

لقد ازداد البايتكود عن السابق لأنَّ الشيفرة البرمجيّة التي أنشِأت سابقًا داخل ملف يدعى `com/packt/chapter5/_5_x_inlineKt$first$1` أصبحت مولّدة `inline`، وفي الحقيقة، لا يوجد صنف إضافي، ويمكنك أن ترى من البايتكود السابق أننا لم نعد نخصّص نسخة كائن.

يجب استخدام هذه الميّزة بعناية، لأن كميّة التعليمات البرمجيّة قد تزداد، ولكن إذا كان ذلك يعني تجنُّب

العديد من التخصيصات (allocations) في حلقة ضيِّقة، فقد يكون العائد جديرًا بالاهتمام، وخاصـة على الأجهـزة الأبطأ مثل الهواتف المحمولة.

| تنبيه | إذا كان المصرِّف يعتقد أن استعمال دوال مباشرة لن يؤدي إلى التحسينات كثيرة فسيرمي تحذيرًا، ويمكنك تعطيل هذا التحذير إذا أردت. |
|---|---|

### أ. التوصيف Noinline

قـد تـرغب في بعض الأحيـان في جعـل دوال محـدَّدة مباشـرة (inline) فقـط، ولأن الدالـة المباشـرة لا يمكن تعيينها إلى متغير (أي الكلمة inline لا يمكن استعمالها مع متغير)، لذلك نستخدم التوصيف `noinline`.

على سـبيل المثـال، لنغيِّر مثـال `withResource` ليقبـل دالـتين، وسـتُطبّق الدالـة الأولى مثـل السـابق، لكن ستُطبّق الثانية بعد إغلاق المورد:

```
inline fun <T : AutoCloseable, U, V> withResource(resource: T,  before:
(T) -> U, noinline after: (U) -> V): V {
   val u = try {
      before(resource)
   } finally {
      resource.close()
   }
   return after(u)
}
```

في حالة استثناء، لن تُستدعَى الدالة الثانية. دعنـا نقـول لأي سـبب كـان، أنَّنـا لا تـرغب في جعـل الدالـة الثانيـة مباشرة، وبإضافة التوصيف `noinline`، ستُغلَّف الدالة في نسخة `FunctionN` مثل العادة، ولن تتأثر الدالـة الأول وستبقى مباشرة.

لنستدعي هذه الدالة باستخدام عداد أحرف محدَّث الذي يرجع الآن الحجم بالكيلوبايت:

```
fun characterCountInKilobytes(filename: String): Int {
```

```
   val input = Files.newInputStream(Paths.get(filename))
   return withResource( input,
{  input.buffered().reader().readText().length }, { it * 1024 } )
}
fun characterCountInKilobytesExpanded(filename: String): Int {
   val input = Files.newInputStream(Paths.get(filename))
   val size = try {
      input.buffered().reader().readText().length
   } finally {
      input.close()
   }
   val fn: (Int) -> Int = { it * 1024 }
   return fn(size)
}
```

بما أنه يمكن أن يكون لاستعمال دوال مباشرة فوائد كبيرة على الأداء، فإن العديد من دوال المكتبة القياسيَّة معرَّفة بواسطة الكلمة المفتاحية `inline`.

## 9. التجريف والتجزيء

يعتبر مفهوم التجريف (Currying) واحد من التقنيات الشائعة في البرمجة الوظيفيَّة، فالتجريف هي عمليّة تحويل دالة تقبل معاملات متعدّدة إلى مجموعة من الدوال، كل منها يقبل دالة واحدة، فلو كان تعريف الدالة `foo` التالي:

```
fun foo(a: String, b: Int) : Boolean
```

فالنسخة المجرَّفة من هذا ستبدو كالتالي:

```
fun foo(a: String): (Int) -> Boolean
```

لاحظ أن النسخة المجرَّفة من `foo` تُرجع دالة ثانية، والتي بدورها، عند استدعاءها مع عدد صحيح (Int) سترجع قيمة منطقيَّة (Boolean) كالسابق.

التجريف هي تقنيَّة مفيدة للسماح للدوال مع معاملات متعددة بالعمل مع دوال تقبل معاملًا واحدًا.

يرتبط التجريف بفكرة التطبيق الجزئي (partial application) أو التجزيء والذي هو عمليَّة تحدد بعض (وليس جميعها) معاملات الدالة مقدمًا، وتُرجع دالة جديدة تقبل المعاملات المفقودة، وهي المعاملات التي أعطيت لتكون ثابتة، وبعبارةٍ أخرى، سينتج التطبيق الجزئي دالة متخصصة من دالة عامة أكثر.

خذ على سبيل المثال هذه الدالة:

```
fun foo(a: Int, b: Boolean, c: Double): Long
```

تطبيق التجزيء في هذا المثال سيكون بتطبيق الدالة مع المعاملين Int و Double، وإرجاعها دالة جديدة على شكل `(Boolean) -> Long`.

تطبيق التجزيء مفيد لسببين على الأقل، الأول عندما تكون بعض المعاملات متاحة في النطاق الحالي لكن ليس في كل النطاقات فيمكننا تطبيق هذه القيم بشكل جزئي ومن ثم مجرد تمريرها إلى دالة ذات رتبة (arity) أقل، وسيتجنب هذا الحاجة لتمرير كل المعاملات والدالة، وثانيًا، على غرار التجريف، يمكننا استخدام التجزيء لتقليل رتبة الدالة لتتناسب مع رتبة أقل لنوع مدخلات دالة أخرى.

### أ. تطبيق التجريف

دعنا نعرض مثالًا على الحالة الثانيّة، لنفترض أن لدينا دالة التي تطبِّق بعض المنطق، تدعى `compute`، تقبل الدالة `compute()` دالة تسجيل سجلات يمكن استخدامها لعرض حالة التقدُّم:

```
fun compute(logger: (String) -> Unit): Unit
```

تقبل دالة تسجيل السجلات سلسلة نصيّة فقط وتفعل شيئًا بها، لا تهتم الدالة `compute` بما ستفعله، بل ستستدعيها فقط، ولنفترض الآن أن لدينا إطار تسجيل السجلات الذي يوفِّر لنا هذه الدالة:

```
fun log(level: Level, appender: Appendable, msg: String): Unit
```

يمكنك استدعاء هذا بطريقة مشابهة لهذه:

```
log(Level.Warn, System.out, "Starting execution")
```

كما ترى، لا تتوافق بصمة الدالة log مع الدالة المقبولة مع التابع `compute()`، لكن إذا استطعنا تطبيقه جزئيًا لإنشاء دالة على شكل `(String) -> Unit`، فسنستعمل. يمكننا القيام بذلك يدويًا عن طريق التغليف في دالة مُجرَّدة:

```
fun compute {
     msg -> log(Level.Warn, Appender.Console, msg)
}
```

يعمل هذا على ما يرام، لكن ألن يكون من الجميل إذا استطعنا القيام بذلك تلقائيًا، خاصةً إذا كنا نتعامل مع معاملات عديدة؟

لسوء الحظ، لا تدعم كوتلن التطبيق الجزئي أو التجريف، لكن اللغة قويّة وتوفّر ميزات كافية يمكننا عبرها توفير دعم بأنفسنا.

### ب. إضافة دعم التجريف

سنعرض الآن مدى سهولة إضافة دعم للتجريف عن طريق نفس مثال السابق لتسجيل السجلات، الخطوة الأول هي تحديد دوال مُوسّعة في FunctionN، والتي ستُرجع دوال تجريف أي ستُجرِّفها إلى دوال أبسط:

```
fun <P1, P2, R> Function2<P1, P2, R>.curried(): (P1) -> (P2) -> R  = {
   p1 -> {
      p2 -> this(p1, p2)
   }
}

fun <P1, P2, P3, R> Function3<P1, P2, P3, R>.curried(): (P1) ->  (P2) ->
(P3) -> R = {
   p1 -> {
      p2 -> {
         p3 -> this(p1, p2, p3)
      }
```

```
    }
}
```

لم نعرض هنا سوى دعم Function2 و Function3، ولإضافة دعم Function4 وغيرها، يمكن بسهولة نسخ السابق وإضافة المعاملات الأخرى. لاحظ أن Function1 لا تحتاج إلى دالة تجريف لأنها بالفعل مُجرَّفة.

ستلاحظ في التطبيق أننا ببساطة نُرجع دوال متداخلة، وفي كل مرة نطبِّق الدالة، سترجع دالة أخرى مع رتبة أقل بمقدار واحد، وسيُلتقط المعامل كمُغلّف.

والآن، لنعطي تعريفنا السابق لدالة تسجيل السجلات:

```
fun logger(level: Level, appender: Appendable, msg: String)
```

يمكننا تجريف هذه ومن ثم تطبيقها جزئيًا عن طريق استدعائها لأول قيمتين:

```
val logger = ::logger.curried()(Level.SEVERE)(System.out)
logger("my message")
```

لاحظ أنَّنا نحتاج أولا للحصول على مرجع الدالة باستخدام `::` قبل استدعاء الدالة `curried()`، ومن ثم سنطبقها مرتين، وسيبدو الشكل المفصَّل (verbose) كالتالي:

```
val logger3: (Level) -> (Appendable) -> (String) -> Unit
= ::logger.curried()
val logger2: (Appendable) -> (String) -> Unit = logger3(Level.SEVERE)
val logger1: (String) -> Unit = logger2(System.out)
logger1("my message")
```

في المثال السابق، ستُضاف الأنواع بشكل صريح لتوضيح شكل الدالة للقارئ في كل خطوة، وفي كل مرة يطبَّق فيها المعامل، يمكنك أن ترى أن رتبة الدالة المُرجعة يقل بمقدار واحد.

## 10. التحفيظ

التحفيظ (Memoization) هي تقنيّة لتسريع استدعاءات الدالة عن طريق التخزين المؤقت وإعادة

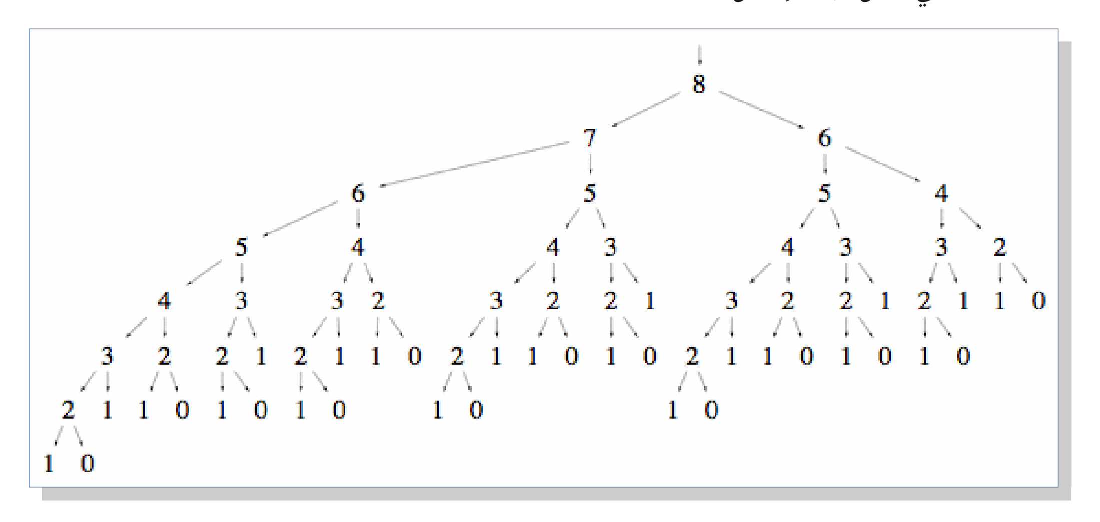
استخدام المخرجات بدلًا من إعادة الحساب لمجموعة معيَّنة من المدخلات، هذه التقنيَّة تقدم مقايضة بين الذاكرة والسرعة، فالتطبيقات المعتادة هي دوال مكلفة حسابيًا أو دوال استدعاء الذاتي التي تتفرَّع من استدعاء دالة نفسها مرات عديدة مع نفس القيم مثل الدالة التي تحسب متتالية فيبوناتشي.

لنستخدم هذا الأخير لاكتشاف أثار التحفيظ، فيمكن تطبيق متتالية فيبوناتشي نفسها بطريقة الاستدعاء الذاتي كالتالي:

```
fun fib(k: Int): Long = when (k) {
   0 -> 1
   1 -> 1
   else -> fib(k - 1) + fib(k - 2)
}
```

لاحظ أنَّه عندما نستدعي `fib(k)`، سنحتاج إلى استدعاء `fib(k-1)` و `fib(k-2)`، ومع ذلك، ستستدعي `fib(k-1)` نفسها `fib(k-2)` و `fib(k-3)`، وهكذا، والنتيجة أننا نجري العديد من الاستدعاءات المتكرِّرة لنفس القيمة، فعلى سبيل المثال، سنستدعي لـ `fib(5)` الدالة `fib(1)` خمسة مرات.

يوضِّح هذا الرسم البياني كيفيَّة زيادة عدد الفروع مع كل مستوى من مستويات فيبوناتشي، حتى الآن معظم الاستدعاءات هي لنفس قيمة الإدخال.



سيتسبب هذا الانفجار في الفروع المتكرّرة في إبطاء الحساب، وكذلك في فيضـان المكـدّس (overflowing the stack) للقيم العليا، وفيما يلي بعض التوقيتات النسبيّة لاستدعاءات فيبوناتشي للقيم المختلفة:

| الدالة | مدة التنفيذ (ميلي ثانية) |
|---|---|
| `fib(5)` | 1 |
| `fib(10)` | 1 |
| `fib(15)` | 1 |
| `fib(20)` | 1 |
| `fib(25)` | 2 |
| `fib(30)` | 5 |
| `fib(35)` | 54 |
| `fib(40)` | 667 |
| `fib(45)` | 6349 |
| `fib(50)` | 69102 |

كما تـرى، بالنسـبة لتعقيـد الـزمن، فـإن فيبوناتشـي هي دالـة أسـيَّة، والسـؤال الـذي يطـرح نفسـه هنـا: أليس من المنطقي أن نخزِّن نتائج `fib` لأي قيمة وإعادة استخدامها في كل مرة تُستدعَى فيه؟ بالتأكيد، ويمكننا إنشاء ذاكـرة تخزين بأنفسنا باستخدام مخزَّن من النوع `map`:

```
val map = mutableMapOf<Int, Long>()
fun memfib(k: Int): Long {
   return map.getOrPut(k) {
      when (k) {
         0 -> 1
         1 -> 1
         else -> memfib(k - 1) + memfib(k - 2)
     }
   }
}
```

الآن، فإن تشغيل التوقيت مرّة أخرى يمنحنا نتائج أفضل بكثير، في الواقع، الفرق ملحوظ، فتكتمل `fib(k)` تقريبًا على الفور لقيم k تصل إلى عدة آلاف (بعد ذلك سنستبدل `Long` أيضًا).

## أ. تنفيذ التحفيظ

السؤال التالي سيكون هل يمكننا جعل هذه العمليّة تلقائيّة لأي دالة؟ الجواب هو نعم، يمكننا ذلك، لكن ليس لدوال الاستدعاء الذاتي، فيمكننا تقديم دالة التحفيظ التي تستخدم قيم الإدخال كمفاتيح في ذاكرة التخزين المؤقّت للبحث عن نتيجة مخزَّنة:

```
fun <A, R> memoize(fn: (A) -> R): (A) -> R {
   val map = ConcurrentHashMap<A, R>()
   return { a ->
      map.getOrPut(a) {
         fn(a)
      }
   }
}
```

لاستخدام الدالة `memoize`، يمكننا ببساطة تمريرها في الدالة الأصليّة، وسنستلم دالة مُغلَّفة تتحقَّق من المخزِّن `map` لوجود أي نتائج جرى حسابها مسبقًا قبل حسابها من جديد، ونحن نستخدم `ConcurrentHashMap` حتى تكون دالة التحفيظ قابلة للاستخدام من خيوط متعددة (multiple threads).

لنفترض أننا أجرينا عمليّة مكلفة، مثلًا إجراء استعلام طويل في قاعدة البيانات، والذي سنسميه query، فيمكننا تغليف الدالة query باستخدام الدالة memoize:

```
val memquery = memoize(::query)
```

لمزيد من التحسين، يمكننا تعريف memoize كدالة مُوسّعة في Function1، مما يسمح لنا باستدعائها باستخدام صياغة النقطة:

```
fun <A, R> Function1<A, R>.memoized(): (A) -> R {
   val map = ConcurrentHashMap<A, R>()
   return {
      a -> map.getOrPut(a) {
         this.invoke(a)
      }
   }
}

val memquery = ::query.memoized()
```

يمكننا إذا اردنا أن نضيف دوال استحضار ذاكرة مشابهة على أصناف FunctionN.

## 11. الأسماء البديلة والمستعارة

قدَّم لنا الإصدار 1.1 من كوتلن ميِّزة جديدة تسمى أنواع بديلة أو مستعارة (type alias) للإشارة إلى أنواع أو تعريفات مفصَّلة، وكما يوحي الاسم، فهو يسمح لنا بالتصريح عن نوع جديد أو شيء ما ليحل مكان نوع أو شيء آخر موجود مسبقًا، ويمكننا فعل ذلك باستخدام الكلمة المفتاحية typealias:

```
typealias Cache = HashMap<String, Boolean>
```

هذه مفيدة لاستبدال بصمات الأنواع المعقّدة (complex type signatures)، فوازن هذين وستعرف أيهما أسهل قراءة، هذه الدالة:

```
fun process(exchange: Exchange<HttpRequest, HttpResponse>):
```

```
Exchange<HttpRequest, HttpResponse>
```

أم هذه الدالة:

```
typealias HttpExchange = Exchange<HttpRequest, HttpResponse>
fun process2(exchange: HttpExchange): HttpExchange
```

ليس للأسماء البديلة أي تأثير سلبي أو إيجابي على الأداء وقت التشغيل، إذ هي عملية استبدال يجريها المصرّف ببساطة، وهذا يعني أنه لن ينشئ شيئًا أو يحجز بايتًا من الذاكرة، لذلك لن يؤثر على الأداء، وهذا يعني أيضًا أنه يمكن استخدام اسميّن مستعاريّن يشيران إلى نفس الشيء بشكل متبادل، فعلى سبيل المثال، تشير جميع هذه التعريفات الثلاثة إلى النوع `String`:

```
typealias String1 = String
typealias String2 = String
fun printString(str: String1): Unit = println(str)

val a: String2 = "I am a String"
printString(a)
```

كما ترى، عرّفنا الدالة لقبول النوع `String1`، والذي هو اسم مستعار للنوع `String`، ولقد تمكّنا من تمريرها إلى `String2` والذي هو أيضًا اسم مستعار للنوع `String`.

ومن عيوبها، أننا لا نستطيع استخدام نوع الأسماء المستعارة لزيادة الأمان على المعاملات من نفس النوع، فعلى سبيل المثال هذا التابع:

```
fun volume(width: Int, length: Int, height: Int): Int
```

إذا غيّرنا هذا لاستخدام نوع الأسماء المستعار لكل بعد من الأبعاد، فلا يزال من الممكن استخدامهم بالتبادل:

```
typealias Width = Int
typealias Length = Int
typealias Height = Int
fun volume(width: Width, length: Length, height: Height): Int
```

نحن قادرون على استدعاء هذه الدالة بأي طريقة من هذه الطرق الخاطئة:

```
val width: Width = 2
val length: Length = 3
val height: Height = 4
volume(width, length, height)
volume(height, width, length)
volume(width, width, width)
```

**تنبيه**

في وقت ترجمة هذا الكتاب، يجب الإعلان على الأسماء المستعارة للنوع في المستوى الأعلى، ويمكنك التأكد من أي تحديث بالإطلاع على القسم Type alias declarations في توثيق Type aliases هذا.

## 12. النوع Either (إمَّا)

في معظم لغات البرمجة الوظيفية، يوجد نوع يسمى `Either` (إمَّا)، ويُستخدم هذا النوع لتمثيل قيمة يمكن أن يكون لها نوعان، ومن الشائع استخدام `Either` لتمثيل قيم نجاح أو فشل، على الرغم من أن هذه ليس الحالة الوحيدة.

على الرغم من أن النوع `Either` لا يأتي ضمن مكتبة كوتلن القياسية، إلا أنه من السهل للغاية إضافته.

لنبدأ بتعريف صنف مجرَّد مغلق (sealed abstract class) مع تطبيقين لكل واحدة من الأنواع التي من الممكن أن يمثلها `Either`:

```
sealed class Either<out L, out R>
class Left<out L>(value: L) : Either<L, Nothing>()
class Right<out R>(value: R) : Either<Nothing, R>()
```

من الدارج إطلاق الاسمين `Left` و`Right` على هذين التنفيذين، وبالاتفاق، عندما يمثِّل الصنف `Either` نجاحًا أو فشلًا، يُستخدَم الصنف `Right` للإشارة لنوع النجاح.

### أ. الطي (Fold)

الدالة الأولى التي سنضيفها إلى Either هي العمليّة fold، وستقبل دالتين، حيث ستطبّق الأولى إذا كان Either هو نسخة للنوع Left، وستطبّق الثانيّة إذا كان Either هو النوع Right، ستُرجع القيمة المُرجعة من أي دالة مطبَّقة:

```
sealed class Either<out L, out R> {
   fun <T> fold(lfn: (L) -> T, rfn: (R) -> T): T = when (this) {
      is Left -> lfn(this.value)
      is Right -> rfn(this.value)
   }
}
```

لنرى كيف يمكن استخدامها، أولًا، لننشئ بعض الأصناف الأساسيَّة والتي سنستخدمها في بقيَّة الأمثلة في هذا القسم:

```
class User(val name: String, val admin: Boolean)
object ServiceAccount
class Address(val town: String, val postcode: String)
```

ولنفترض أن لدينا دالة تُرجِع لنا المستخدم الحالي ودالة أخرى تُرجع لنا عناوين مستخدم معيَّن:

```
fun getCurrentUser(): Either<ServiceAccount, User> = ...
fun getUserAddresses(user: User): List<Address> = …
```

لاحظ أن الدالة getCurrentUser تُرجع لنا Either، والذي يحتوي نوعيّن من المستخدم، الأول هو المستخدم العادي، والثاني هو مستخدم ServiceAccount الخاص، ويمكننا بعد ذلك استخدام Either للحصول على عناوين المستخدم:

```
val addresses = getCurrentUser().fold({ emptyList<Address>() },
{ getUserAddresses(it) })
```

كما ترون، نحن نتعامل مع البحث بالاعتماد على النوع الذي قدمناه، في هذه الحالة، لا يحتوي حساب الخدمة على أية عناوين، ولذلك سنعرض قائمة فارغة فقط.

### ب. الإسقاط

من الشائع أن نرى عمل على `Either` يسمح لنا بربط (map) وترشيح (filter) وجلب قيمة ...إلخ. وهذه العمليات معرَّفة لذلك يمكنك تطبيقها على أحد الأنواع فقط، ولا تعمل في حالة الأخرى، ويسمى هذا بالإسقاط الأيمن ( right projection) أو الأيسر ( left projection).

سيقرّر المستخدم ما إذا كانوا مهتمين بالحالات على اليسار أو اليمين، و عن طريق استدعاء الدالة، سيحصلون على إسقاط يحتوي على القيمة التي تهمهم، أو بدون قيمة إذا كان النوع الذي يرغبون به ليس من الأنواع التي يحتويها `Either`.

الطريقة التي سنختارها لتنفيذ ذلك هي إنشاء صنفي إسقاط `ValueProjection` و `EmptyProjection`، فسيُنفِّذ الصنف `ValueProjection` الدوال، وأما `EmptyProjection` فسيُنفِّذ عمليات فارغة أو واهية (no-ops). سيحتوي الصنف `Either` على دوال للحصول على إسقاط لأي جانب مطلوب.

لنبدأ بإنشاء الصنف `Projection` المجرَّد، والذي سيُعرِّف الدوال التي نحن مهتمين بها والذي سيكون النوع الأعلى (supertype) لصنفي التنفيذ:

```
sealed class Projection<out T> {
  abstract fun <U> map(fn: (T) -> U): Projection<U>
  abstract fun getOrElse(or: () -> T): T
}
```

سنبدأ الآن بدالتيّن: الأولى `map`، والتي ستحوّل القيمة إذا كان الإسقاط هو ما نرغب به والثانية `getOrElse` التي ستُرجع القيمة أو ستطبّق الدالة الافتراضيّة، الخطوة القادمة هي تنفيذ ذلك لكلا الصنفين:

```
class ValueProjection<out T>(val value: T) : Projection<T>() {
  override fun <U> map(fn: (T) -> U): Projection<U> =
```

```
ValueProjection(fn(value))
   override fun getOrElse(or: () -> T): T = value
}

class EmptyProjection<out T> : Projection<T>() {
   override fun <U> map(fn: (T) -> U): Projection<U> =
EmptyProjection<U>()
   override fun getOrElse(or: () -> T): T = or()
}

fun <T> Projection<T>.getOrElse(or: () -> T): T = when (this) {
   is EmptyProjection -> or()
   is ValueProjection -> this.value
}
```

لاحظ أنَّ EmptyProjection ستُرجع نسخة أخرى لـ EmptyProjection فقط دون ربط (mapping) أي شيء، فـ ValueProjection هي من تنفِّذ العملية.

**تنبيه**

تطبّق getOrElse كدالة مُوسّعة في Projection نفسه لأن بصمة الدالة تتطلّب أن T هو خرج ممثَّل بالدالة or، ويقطع هذا التباين المشترك ما لم نستخدم دالة مُوسّعة. سنتحدث عن التباين في فصل لاحق.

الخطوة الأخيرة هي تحديث الصنف Either لإرجاع هذه الإسقاطات عند الطلب:

```
sealed class Either<out L, out R> {

   fun <T> fold(lfn: (L) -> T, rfn: (R) -> T): T = when (this) {
      is Left -> lfn(this.value)
      is Right -> rfn(this.value)
   }

   fun leftProjection(): Projection<L> = when (this) {
```

```
        is Left -> ValueProjection(this.value)
        is Right -> EmptyProjection<L>()
    }

    fun rightProjection(): Projection<R> = when (this) {
        is Left -> EmptyProjection<R>()
        is Right -> ValueProjection(this.value)
    }
}
```

يمكننا الآن استخدام هذا على النحو التالي:

```
val postcodes = getCurrentUser().rightProjection()
.map { getUserAddresses(it) }
.map { addresses.map { it.postcode } }
.getOrElse { emptyList() }
```

هذا تابع مشابه للمثال السابق، لكن لاحظ كيف يمكننا متابعة `map` على النتائج، ومن ثم تطبيق الافتراضي في النهائية، إذا أرجعت `Either` قيمة ليست `Right`، فلن يكون لها أية تأثيرات.

## ت. المزيد من دوال الإسقاط

سنستمر بإضافة المزيد من دوال الإسقاط وهي `exists` و `filter` و `toList` و `orNull`.

ستقبل `exists` دالة، وإذا كان الإسقاط يملك قيمة، فستطبَّق عليه الدالة وسترجع القيمة المنطقيّة (Boolean) للنتيجة، وسيُرجع `false` إذا كان الإسقاط فارغًا:

```
abstract fun exists(fn: (T) -> Boolean): Boolean
```

كما يوحي الاسم، سيجري `filter` عمليّة ترشيح على الإسقاط، ستُطبِق قيمة الإسقاط الدالة وسترجع إسقاطًا فارغًا إذا كانت نتيجة دالة `filter` قيمة خطأ (false):

```
abstract fun filter(fn: (T) -> Boolean): Projection<T>
```

ستُرجع الدالة toList قائمة بالقيم أو قائمة فارغة إذا كان الإسقاط فارغ:

```
abstract fun toList(): List<T>
```

وأخيرًا، ستُرجِع orNull القيمة أو null إذا كان الإسقاط فارغًا:

```
abstract fun orNull(): T?
```

سنضيف بعض الدوال إلى النوع Either لتسمح لنا بفحص النوع، لذا سيبدو التصميم النهائي للنوع Either الأساسي كالتالي:

```
sealed class Either<out L, out R> {
  fun <T> fold(lfn: (L) -> T, rfn: (R) -> T): T = when (this) {
    is Left -> lfn(this.value)
    is Right -> rfn(this.value)
  }

  fun leftProjection(): Projection<L> = when (this) {
   is Left -> ValueProjection(this.value)
   is Right -> EmptyProjection<L>()
  }

  fun isLeft() = when (this) {
    is Left -> true
    is Right -> false
  }

  fun rightProjection(): Projection<R> = when (this) {
   is Left -> EmptyProjection<R>()
   is Right -> ValueProjection(this.value)
  }

  fun isRight() = when (this) {
```

```
        is Left -> false
        is Right -> true
    }

}
```

مع الأنواع الفرعيّة التالية التي تنفذ كلا الحالتين:

```
class Left<out L>(val value: L) : Either<L, Nothing>()
class Right<out R>(val value: R) : Either<Nothing, R>()
```

ودالة مُوسّعة مطلوبة:

```
fun <T> Projection<T>.getOrElse(or: () -> T): T = when (this) {
    is EmptyProjection -> or()
    is ValueProjection -> this.value
}

sealed class Projection<out T> {
    abstract fun <U> map(fn: (T) -> U): Projection<U>
    abstract fun exists(fn: (T) -> Boolean): Boolean
    abstract fun filter(fn: (T) -> Boolean): Projection<T>
    abstract fun toList(): List<T>
    abstract fun orNull(): T?
}

class EmptyProjection<out T> : Projection<T>() {
    override fun <U> map(fn: (T) -> U): Projection<U> =
EmptyProjection<U>()
    override fun exists(fn: (T) -> Boolean): Boolean = false
    override fun filter(fn: (T) -> Boolean): Projection<T> = this
    override fun toList(): List<T> = emptyList()
    override fun orNull(): T? = null
}
```

```
class ValueProjection<out T>(val value: T) : Projection<T>() {
   override fun <U> map(fn: (T) -> U): Projection<U> =
ValueProjection(fn(value))
   override fun exists(fn: (T) -> Boolean): Boolean = fn(value)
   override fun filter(fn: (T) -> Boolean): Projection<T> = when
(fn(value)) {
      true -> this
      false -> EmptyProjection()
   }

   override fun toList(): List<T> = listOf(value)
   override fun orNull(): T? = value
}
```

يمكننا الآن تنفيذ الشيفرة المصدريّة كالتالي:

```
val service: ServiceAccount? = getCurrentUser().leftProjection().orNull()
val usersWithMultipleAddresses = getCurrentUser().rightProjection()
.filter { getUserAddresses(it).size > 1 }
val isAdmin = getCurrentUser().rightProjection().exists { it.admin }
```

## 13. تخصيص اللغات مخصَّصة المجال

اللغة مخصَّصة المجال (domain-specific language وتختصر إلى DSL)، هي لغة مخصَّصة لمجال معيَّن، فعلى سبيل المثال، غالبًا ما تأتي برامج تعقّب المشكلات عبر الإنترنت مثل Jira بـ "لغة صغيرة" للاستعلام، مصمَّمة لتسهيل إجراء عمليات البحث المتقدِّمة.

في البرمجة، نرى في أغلب الأحيان لغات مخصَّصة المجال في شكل واجهة تطبيقات API مصمَّمة لجعل استخدام تلك الواجهة أسهل.

وبما أن كوتلن يوفِّر العديد من المميزات حول استخدام معاملات الدوال المسماة (functions-named parameters)، و المعاملات الافتراضيّة، وزيادة تحميل العامل، ودوال infix مما يجعل كوتلن لغة قويَّة لإنشاء

لغة مخصَّصة لأي مجال تريده.

في هذا القسم، سننشئ لغة مخصَّصة لاستخدامه في التوكيدات (assertions)، ويُستخدَم هذا النوع من الوظائف في الاختبار أو تطوير السلوك، وفي الواقع، سنخصِّص فصلًا كاملًا للاختبار في وقت لاحق من هذا الكتاب باستخدام مكتبة KotlinTest المتقدِّمة.

### أ. دوال التدوين infix ككلمات مفتاحية

سنكتب توكيدًا بسيطًا للتأكد من تساوي قيمة لقيمة أخرى، ويمكننا القيام بذلك من خلال إنشاء دالة للتحقق من المساواة:

```
fun equals(first: Any, second: Any): Unit {
     if (first != second)
            throw RuntimeException("$first was not equal to $second")
}
```

ويمكننا بعد ذلك استخدام هذه الدالة على النحو التالي:

```
equals("foobar", "foobaz")
```

هذا جيِّد، لكن ليس محددًا بنطاق، وستكون الخطوة التالية جعل هذه الدالة من النوع `infix`:

```
infix fun Any.equals(other: Any): Unit {
     if (first != second)
            throw RuntimeException("$first was not equal to $second")
}
```

لاحظ أنَّ الدالة أصبحت الآن دالة مُوسَّعة للسماح بعامل `infix`، ويمكننا استخدامه على النحو التالي:

```
"foobar" equals "foobaz"
```

هذا أفضل بقليل، وربما يمكننا إعادة تسميّة الدالة لجعلها أكثر وضوحًا للقارئ:

```
fun Any.shouldEqual(other: Any): Unit {
     if (this != other)
            throw RuntimeException("$this was not equal to $other")
}
```

والآن، سيُصبِح تأكيدنا كالتالي:

```
"foobar" shouldEqual "foobaz"
```

يمكننا الآن بناء توكيدات أخرى للحالات غير المتساوية، فيمكننا على سبيل المثال التأكد من أن مجموعة ما تحتوي على عنصر معيَّن:

```
listOfNames.contains("george") shouldEqual true
```

لكن أليس من أفضل إذا جعلنا التوكيد يهتم بالشيفرات البرمجيّة المتكرِّرة حتى يمكننا من كتابة شيء أكثر قابليّة للقراءة قليلًا؟ سيكون مثاليًا إذا استطعنا كتابة شيء مثل هذا:

```
listOfNames shouldContain "george"
```

يمكننا فعل ذلك عن طريق إنشاء كلمة مفتاحية أخرى في شكل دالة مُوسّعة:

```
infix fun <E> Collection<E>.shouldContain(element: E): Unit {
     if (!this.contains(element))
            throw RuntimeException("Collection did not contain $element")
}
```

لاحظ أنَّ ذلك سيعمل على أي نوع من التجميعات، كما أن لديه فائدة إضافيّة في أن المصرِّف سيتحقَّق ما إذا كان نوع العنصر هو نفسه نوع المجموعة، لذا لن نتمكَّن من تصريف شيفرة برمجيّة مشابهة لهذه:

```
listOfNames shouldContain 10.0
```

لنطوِّر هذا قليلًا بإضافة دوال تسمح لنا بالجمع بين عدة توكيدات، فالهدف الأساسي هو القدرة على كتابة شيفرات برمجيّة مشابهة لهذه:

```
listOfNames shouldContain "george" or listOfNames should beEmpty()
```

نعلم أنّنا سنحتاج إلى دالة من النوع infix باسم or، والتي تجمع بين توكيدين، ويجب أن تكون دالة مُوسّعة أو دالة تابعة حتى تتمكن من استخدام infix؛ الفكرة الأولى هي تعريف or في Unit:

```
infix fun Unit.or(other: Unit): Unit
```

ومع ذلك، بما أن التوكيدات ترمي استثناء، فإن الجانب الأيسر يمكن أن يكون بالفعل قد رمى استثناء قبل استدعاء or، أي لا يمكننا التقاطه، وفي هذه الحالة، نحن بحاجة إلى استدعاء التوكيدات بعد أن تُجمَع، وفي نفس الوقت، هل يمكننا تجنب تكرار الجانب الأيسر المكرَّر؟

لنعرِّف بعض الأنواع، مثل Matcher، والذي سيلتقط التوكيد ويسمح لنا بالفصل (عبر or) والاقتران (عبر and):

```
interface Matcher<T> {
     fun test(lhs: T): Unit
}
```

الفكرة هنا أنّنا سنقوم بطريقة ما بإنشاء مطابقات باستخدام الكلمات المفتاحية، وستُستدعَى هذه المطابقات لتشغيل الاختبارات.

في البداية، سنحتاج إلى تنفيذ Matcher لكل من contains و empty:

```
fun <T> contain(rhs: T) = object : Matcher<Collection<T>> {
   override fun test(lhs: Collection<T>): Unit {
      if (!lhs.contains(rhs))
       throw RuntimeException("Collection did not contain $rhs")
   }
}

fun <T> beEmpty() = object : Matcher<Collection<T>> {
   override fun test(lhs: Collection<T>) {
      if (lhs.isNotEmpty())
```

```
        throw RuntimeException("Collection should be empty")
    }
}
```

نحتاج الآن إلى طريقة لاستدعاء هذه على المتلقي. دعنا نقدم دالة تدعى `should` والتي ستفعل هذا من أجلنا:

```
infix fun <T> T.should(matcher: Matcher<T>) {
    matcher.test(this)
}
```

كما ترى، دالة `should` هي مجرّد أداة لتفعيل المطابقات، ولذلك يمكن إعادة كتابة المثال السابق كالتالي:

```
listOfNames should contain("george")
```

حان الآن وقت إضافة الدالة `or` للجمع بين المطابقات، وكما ذكرنا في السابق، يجب أن تكون هذه دالة مُوسّعة، لذا سنضيفها إلى الواجهة `Matcher`:

```
interface Matcher<T> {

  fun test(lhs: T): Unit

  infix fun or(other: Matcher<T>): Matcher<T> = object : Matcher<T> {
    override fun test(lhs: T) {
      try {
        this@Matcher.test(lhs)
      } catch (e: RuntimeException) {
        other.test(lhs)
      }
    }
  }
}
```

لاحظ أنّنا نحتاج إلى نجاح أحد المطابقات لـ `or` لتكون صحيحة، لذا يجب التقاط أي استثناء يُرمَى من قبل

المطابق الأول لإعطاء المطابق الثاني فرصة للعمل، وعلى العكس، إذا نجح المطابق الأول، فلا داعي لاستدعاء الثاني على الإطلاق.

يتيح لنا جمع كل هذا معًا بإنشاء صياغة تحقق هدفنا الأصلي:

```
listOfNames should (contain("george") or beEmpty())
```

دعنا نستفيد من مستقبلات دالة كوتلن للسماح لنا بكتابة دالة يمكننا استخدامها لعمل توكيدات عديدة دفعة واحدة:

```
listOfNames should {
      contain("george")
      beEmpty()
}
```

## ب. استخدام مستقبلات الدالة في DSL

يمكن استخدام متلقيات الدوال بطريقة قويَّة عند كتابة لغة مخصَّصة المجال DSL، فهم يسمحون لنا بتقديم توابع يمكن استخدامها في دالة مجرَّدة، لكن يقتصر استخدامها في "القسم" المناسب.

فعلى سبيل المثال، لنقدّم بعض المطابقات التي تعمل فقط على التجميعات، وتسمح للعديد منها بالعمل في نفس الوقت، الفكرة هي عمل صياغة مثل هذه:

```
listOfNames should {
      contain("george")
      contain("harry")
      notContain("francois")
      haveSizeLessThan(4)
}
```

ستُعرَّف التوكيدات `contain` و `notContain` و `haveSizeLessThan` في صنف واحد، والذي سيمثِّل المستقبل للدالة المجرَّدة، وسيسمح هذا باستدعاء هذه الدالة دون الحاجة إلى بادئة:

```
class CollectionMatchers<T>(val collection: Collection<T>) {
```

```
    fun contain(rhs: T): Unit {
        if (!collection.contains(rhs))
            throw RuntimeException("Collection did not contain $rhs")
    }

    fun notContain(rhs: T): Unit {
        if (collection.contains(rhs))
            throw RuntimeException("Collection should not contain $rhs")
    }

    fun haveSizeLessThan(size: Int): Unit {
        if (collection.size >= size)
            throw RuntimeException("Collection should have size less  than
$size")
    }
}
```

قمنا الآن بتعريف التوكيدات في صنف يـدعى `CollectionMatchers`، وسنحتاج إلى دالـة تستخدم هـذا كمسـتقبل لكتلـة من الشـيفرات البرمجيَّة، لننشـئ دالـة أخـرى باسـم `should` لفعل هذا:

```
infix fun <T> Collection<T>.should(fn: CollectionMatchers<T>.() ->  Unit)
{
    val matchers = CollectionMatchers(this)
    matchers.fn()
}
```

كما ترى، استعملنا الكلمة المفتاحية `infix` مع الدالة مرةً أخرى، على الرغم من ذلك، فإنَّ الجـزء الرئيسـي هـو الدالة التي تحتوي على مجموعة المستقبل.

أنشأنا داخل جسم الدالة نسخة للصنف `CollectionMatchers` واستدعينا الدالة المزوَّدة عليها، والنتيجـة النهائيَّة هي دعم الصـيغة المطلوبـة، ولأنَّ دوال `contains` و `notContains` و `haveSizeLessThan` هي

دوال أعضاء للصنف `CollectionMatchers`، لا يمكننا استدعاء تلك الوظائف في المكان الخاطئ.

## 14. التحقق من الأخطاء وتراكمها

سنغطي في ختام مقدمتنا البرمجة الوظيفيّة نمط شائع آخر وهو تراكم الأخطاء (error accumulation)، ويشار إلى هذا في بعض الأحيان بالتحقق (validation).

الفكرة هي أن لدينا سلسلة من دوال تتحقق بشكل منفرد من خطأ في القيمة، ويمكنهم إرجاع نوع من القيمة الناجحة إذا كان المدخل صحيح، ونوع من القيمة الخاطئة إذا كان المُدخل مختلف أو خطأ، وستُجمَع هذه الدوال الفرديَّة، مع الاحتفاظ بجميع الأخطاء (إن وُجدت)، وفي النهاية، يمكننا الاطلاع على تلك الأخطاء المتراكمة جميعها.

لنبدأ بوضع نماذج للقيم الصحيحة والخطأ التي يمكننا استخدامها، وسنسمها `Valid` و `Invalid` على التوالي، وسيكون كلاهما امتداد لصنف أعلى يسمى `Validation`:

```
sealed class Validation
object Valid : Validation()
class Invalid(val errors: List<String>) : Validation()
```

لاحظ أنَّ حالة `Invalid` تحتوي على قائمة من الأخطاء على شكل سلاسل نصيَّة، وسيُضاف كل خطأ إليها، ويسمى هذا بتراكم الأخطاء، وأما `Valid` فهو مجرَّد كائن لا يحمل أي حالة.

سيكون مثالنا هو التحقُّق من صحَّة النسخة `Student`، وإليك الصنف `Student`:

```
class Student(val name: String, val studentNumber: String, val  email:
String)
```

سنحتاج إلى بعض الدوال التي تتأكد من أن المعاملات `name` و `studentNumber` و `email` صحيحة:

```
fun isValidName(name: String): Validation {
   return if (name.trim().length > 2)
      Valid
   else
      Invalid("Name $name is too short")
}
```

```
fun isValidStudentNumber(studentNumber: String): Validation {
   return if (studentNumber.all { Character.isDigit(it) })
      Valid)
   else
      Invalid("Student number must be only digits: $studentNumber")
}

fun isValidEmailAddress(email: String): Validation {
   return if (email.contains("@"))
      Valid
   else
      Invalid("Email must contain an '@' symbol")
}
```

جميع الدوال واضحة، النقطة الأساسيّة هي أنّها تعيد نسخة من `Validation`: إما `Valid` أو `Invalid` بناءً على نتيجة التحقق من الخطأ، وبالطبع فحص البريد الإلكتروني بسيط للغاية وذلك لأنّ هذا المقال يتعلق بتراكم الأخطاء وليس كتابة تحقيق للبريد الإلكتروني بشكل صحيح.

سنضيف تابع مساعد لكائن مرافق `invalid` حتى نتمكن من إنشاء نسخة من قيمة سلسلة نصيّة منفردة، ولتجنّب بعض التكرارات:

```
class Invalid(val errors: List<String>) : Validation<Nothing>() {
   companion object {
      operator fun invoke(error: String) = Invalid(listOf(error))
   }
}
```

سنحتاج الآن إلى طريقة لتراكم القيم والأخطاء معًا، وسيكون من الرائع إذا استطعنا فعل هذا عن طريق بعض العوامل للحفاظ على الشيفرة البرمجيّة قابلة للقراءة، لذلك لنعيد تعريف استخدام `plus`:

```
sealed class Validation {
```

```
   abstract infix operator fun plus(other: Validation): Validation
}
```

سيحتاج كل صنف فرعي للصنف `Validation` إلى تنفيذ التالي:

```
class Invalid(val errors: List<String>) : Validation() {

   override fun plus(other: Validation): Validation = when (other)  {
      is Invalid -> Invalid(this.errors + other.errors)
      is Valid -> this
   }
}
object Valid : Validation() {
   override fun plus(other: Validation): Validation = when (other)  {
      is Invalid -> other
      is Valid -> this
   }
}
```

نحن الآن قادرون على دمج أمثلة `Validation` معًا:

```
val validation = isValidName(student.name) +
isValidStudentNumber(student.studentNumber) +
isValidEmailAddress(student.email)
```

وأخيرًا، نريد أن نفعل شيئًا مفيدًا مع الأخطاء، بالطبع، يمكننا فقط الوصول إليه مباشرةً كحقل ولكن دعنا نضيف دالة مساعدة للسماح لنا بالحصول على قيمة، أو تطبيق شيء بشكل افتراضي.

ستحمل هذه الدالة البصمة التالية:

```
abstract fun <T> getOrElse(t: T, or: (List<String>) -> T): T
```

سيُنفَّذ هذا في كل الأصناف الفرعية كالتالي:

```
class Invalid(val errors: List<String>) : Validation() {
```

```
   override fun <T> getOrElse(t: T, or: (List<String>) -> T): T =
or(errors)
}

object Valid : Validation() {
   override fun <T> getOrElse(t: T, or: (List<String>) -> T): T = t
}
```

يمكننا الآن استخدام نتائج خطوة التحقق كالتالي:

```
fun validateStudent(student: Student): Student {

val validation = isValidName(student.name) +
isValidStudentNumber(student.studentNumber) +
isValidEmailAddress(student.email)
return validation.getOrElse(student, {
   throw  RuntimeException("Error creating student. The errors are $it") }
)
}
```

| تنبيه | هنالك اختلافات كثيرة لهذا التابع، ومن الاختلافات الشائعة هو تراكم القيم مع الأخطاء جنبًا إلى جنب، ومن ثم استخدام هذه القيم في دالة تحويل لإرجاع كائن باني نهائي. سنتحدث عن التباين في فصل لاحق. |
| --- | --- |

## 15. خلاصة الفصل

ناقشنا في هذا الفصل الحالات الأكثر استخدامًا للدوال، وخاصة دوال المستوى الأعلى التي تدعم مكتبة التجميعات (collection) في معظم اللغات الحديثة، وكوتلن ليست استثناء من هذا، رأينا كيف أن العديد من الميزات التي يوفرها كوتلن للدوال يمكن الاستفادة منها لكتابة أجزاء من اللغة مخصَّصة المجال DSL، وأخيرًا، قدمنا تعابير مشتركة في فضاء البرمجية الوظيفية، كما تطرَّقنا إلى النوع `Either` وعمليات التحقق (Validation).

في الفصل القادم، سنناقش المواضيع المتمِّمة للدوال (الخاصيات [properties]) والتي تُستخدَم لاسترداد وتحديث القيم في الكائنات.

الفصل السادس:

# الخاصيات



لقد تطرَّقنا إلى الخاصيات (Properties) باختصار في الفصل الثالث، البرمجة كائنيَّة التوجُّه في كوتلن، وسنلقي في هذا الفصل نظرة مفصَّلة عليها، وسنتعلم عن:

- الخاصيات العامة (General properties)
- المرئية (Visibility)
- التهيئة الكسولة (Lazy initialized) واللاحقة (Late initialized)
- الخاصيات المُعمَّمة (Delegated properties)
- متى تستخدم الخاصيات بدلًا من التوابع

وعلاوةً على ذلك، سترى كيف تستخدم خاصيَّة كوتلن من جافا وسنلقي نظرة على البايتكود الناتج لفهم ما يفعله المصرِّف البرمجي، وإذا كنت معتادًا على #C، ستكون المعلومات الموجودة هنا مألوفة، فلقد جُلب مفهوم الخاصيات من عالم NET..

## 1. لماذا نستخدم الخاصيات؟

الخاصيات ليست أكثر من شيء يسمح لشيفرتك البرمجيَّة باستخدام صياغة مبسَّطة، فتدعم كوتلن الخاصيات البسيطة (simple properties) والخاصيات المُعمَّمة (ستعرف لاحقًا ما هي).

كم مرةً كتبت صنفًا يحتوي على معلومات الحالة التي يمكن أن تسترجعها أو تُغيرها؟ عادةً، تأتي معلومات الحالة في شكل حقول، وهذا صنف نموذجي يحدد حقلين:

```
class Student {
     private val name:String;
     private val age:Int;
}
```

تُكتب أصناف مثل هذه في جافا بشكل متكرِّر (لحسن الحظ أن IntelliJ قوي في توليد التعليمات البرمجيّة وإعادة صياغتها)، إذ سيكون هنالك في العادة تابعان لكل حقل: الجالب getter والضابط setter وستشبه الشيفرة البرمجيّة هذه:

```
public class Student {
   private String name;
   private intage;
   public Student(String name, intage){
      this.name= name;
      this.age= age;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name= name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(intage) {
      this.age= age;
   }
}
```

دعنا الآن نرى كيف يمكننا كتابة الشيفرة البرمجيّة السابقة في كوتلن:

```
class Student(name: String, age: Int) {
   public var Name = ""
   set(value) {
      field = value
   }

   public var Age = 20
   set(value) {
      field = value
```

```
    }

    init {
        Name = name
        Age = age
    }
}
```

جميلٌ! لاحظ أنه يجب تعريف كتلة `init` بعد تعريفات الخاصيات، وهذه من السلبيات التي نرجو أن تتغيَّر في المستقبل، وإذا كتبت كتلة `init` بعد تعريفات الخاصيات فستحصل على خطأ التصريف، وهذا مثال لاستخدام الصنف السابق:

```
val student = Student("Jamie Fox", 20)
print("${student.Name} is ${student.Age} years old")
student.Age+=1
print("${student.Name} is ${student.Age} years old")
```

يبدو هذا سهلًا، لكن دعنا نرى ما يحدث في الواقع تحت الغطاء، سنعود مرّةً أخرى إلى أداة `javap` للحصول على بايتكود منتج وتشغيل سطر الأوامر، وستحصل على شيء مماثل لما يلي:

```
public final class com.programming.kotlin.chapter06.Student {
    public final java.lang.String getName();
    Code:
        0: aload_0
        1: getfield #11 // Field Name:Ljava/lang/String;
        4: areturn
    public final void setName(java.lang.String);
    Code:
        ...
        6: aload_0
        7: aload_1
        8: putfield #11 // Field Name:Ljava/lang/String;
```

```
      11: return
      ...
   public com.programming.kotlin.chapter06.Student(java.lang.String, int);
   Code:
      ...
      24: invokevirtual #42 // Method setName:(Ljava/lang/String;)V
      27: aload_0
      28: iload_2
      29: invokevirtual #44 // Method setAge:(I)V
      32: return
 }
```

من أجل البساطة، لقد تركت جزءًا من الشيفرة البرمجية، سيُوضح المقتطف ما الذي فعله المصرّف، فسيولّد تابعي الجلب get والضبط set[10] فضلًا عن مجموعة حقول مساعدة (backing field) من أجل name و age، هل لاحظت استخدام الكلمة المفتاحية field داخل كتلة set؟ فهو اسم مستعار لدعم الحقل المساعد المولّد من أجلنا، وإذا استخدمت شيفرة برمجيّة لكوتلن من جافا، سينتهي بك المطاف مع نمط نموذجي لاستدعاء get*** و set***:

```
Student student = new Student("Alex Wood", 20);
System.out.println("Student " + student.getName() + " is " +
student.getAge() + " years old");
student.setAge(student.getAge() + 1);
```

يمكننا الاستفادة الكاملة من قدرات المصرّف عندما يتعلّق الأمر بخاصيات بسيطة، إذا عرّفنا معاملات الباني Student على أنّها val، فسنترك المصرّف ليقوم بكل العمل المطلوب من أجلنا لأننا سنولّد جالبًا getter وليس ضابطًا setter.

في العادة، في حالة Student، سترغب بتوفير ضابط setter مخصّص لأنك تريد فرض تحقيقات للصحّة، فعلى سبيل المثال، يجب رمي استثناء عند وضع عمر أصغر من 1، لكن سنترك هذا لاحقًا:

10 التابع الجالب أو getter هو من يجلب قيمة الخاصية، والتابع الضابط أو setter هو من يضبط قيمة الخاصية.

```
class Student(var name: String, var age: Int)
```

إذا نظـرت إلى بـايتكود البـاني، سـتلاحظ نفس الشـيء تقريبـًا، الفـرق في هيكـل الجسـم، فبـدلًا من اسـتدعاء ( set*** #** invokevirtual فسيستخدم putfield فقط لضبط قيم الحقل المساعد.

## 2. الصياغة والاختلافات

صياغة إعلان خاصيَّة هو التالي:

```
var/val<propertyName>:<PropertyType>[=<property_initializer>]
   [<getter>]
   [<setter>]
```

كل من جزء من initializer و setter اختياري، وعلاوةً على ذلك، يمكن أن يُترَك نـوع الخاصـيَّة بمـا أن المصرّف يمكن أن يستنتجه، مما يوفِّر لك نقرات على لوحة المفاتيح، ومع ذلك، من المستحسن إضافة نوع الخاصيَّة من أجل وضوح الشيفرة البرمجيَّة.

إذا عرَّفت خاصيَّة للقراءة فقط باستخدام الكلمة المفتاحيـة val، فستحصـل على جـالب getter فقـط بـدون ضابط setter، فتخيَّل أن عليك تحديد التسلسل الهرمي لصـنف لتطـبيق الرسـم، فسـتحتاج إلى خاصـيَّة للمسـاحة، وهذا هو نموذج لمثل هذه الخاصيَّة عندما يتعلق الأمر بصنف Rectangle:

```
interface Shape {
   val Area: Double
   get;
}

class Rectangle(val width: Double, val height: Double) : Shape {
   override val Area: Double
   get() = width * height

   val isSquare: Boolean = width == height
}
```

يطبِّق صنف rectangle واجهة Shape ولذلك فعليه تعريف خاصيَّة Area، وبصرف النظر عن ذلك، فإنه يضيف خاصيَّة جديدة للتحقق مما إذا كان المستطيل هو في الواقع مربّع، وقد تعتقد أنه يجب علينا جلب حقلٍ مساعدٍ للخاصيَّة Area ومع ذلك فإننا لا نحتاج إلى ذلك، وهذا ما يكشفه البايتكود:

```
public final class com.programming.kotlin.chapter06.Rectangle implements
com.programming.kotlin.chapter06.Shape {
   public double getArea();
      Code:
       0: aload_0
       1: getfield      #12                  // Field width:D
       4: aload_0
       5: getfield      #15                  // Field height:D
       8: dmul
       9: dreturn
public final booleanisSquare();
      Code:
       0: aload_0
       1: getfield      #12                  // Field width:D
       4: aload_0
       5: getfield      #15                  // Field height:D
       16: iconst_0
     17: ireturn
public final double getWidth();
   Code:
      0: aload_0
      1: getfield      #12                  // Field width:D
   4: dreturn
public final double getHeight();
   Code:
   0: aload_0
   1: getfield      #15                  // Field height:D
     4: dreturn
```

```
publiccom.programming.kotlin.chapter06.Rectangle(double, double);
   Code:
      6: putfield       #12                  // Field width:D
      11: putfield       #15                  // Field height:D
     14: return
}
```

في بعض الأحيان، لا تكون الشيفرة البرمجيَّة للجالب getter بسيطة مثل إرجاع قيمة الحقل المساعد فقط، ففي مثل هذه الحالة، ستحتاج إلى توفير الحقل المساعد بنفسك، ويجب عليك اتباع أفضل ممارسات وتجنّب وجود منطق معقّد في الجالب الخاص بك، فتخيَّل أن لدينا صنفًا يوفِّر مجموعة من الكلمات الرئيسيَّة، ونريد تهيئة الحقل بتكاسل في أول استخدام، وعندما نتحدَّث عن خاصيات مُعمَّمة، سنرى نهجًا مختلفًا عندما تكتب شيفرة كوتلن اصطلاحيَّة، وهذا مثال لكيفيَّة تنفيذ تخزين في الذاكرة المؤقتة للكلمات الرئيسية (keywords) في الوقت الحالي:

```
class Lookup {
   private var _keywords: HashSet<String>? = null

   val keywords: Iterable<String>
   get() {
      if (_keywords == null) {
         _keywords = HashSet<String>()
      }
      return _keywords ?: throw RuntimeException("Invalid keywords")
   }
}
```

## 3. المرئية

تنطبق قواعد الوصول إلى الرؤية التي ناقشناها للحقول على الخاصيات أيضًا، ولذلك، يمكن أن يكون لديك خاصيات خاصة أو محميّة أو عامة (بشكل افتراضي)، وعلاوةً على ذلك، فإنه قد تملك ضابطًا setter مختلف المرئية وأكثر تقييدًا من الجالب getter (تولِّد الشيفرة البرمجيَّة للجالب getter بشكل تلقائي في الحالة التاليَّة):

```
class WithPrivateSetter(property: Int) {
   var SomeProperty: Int = 0
      private set(value) {
         field = value
      }

   init {
      SomeProperty = property
   }
}

val withPrivateSetter = WithPrivateSetter(10)
println("withPrivateSetter:${withPrivateSetter.SomeProperty}")
```

هنالك سيناريوهات تخضع فيها الخاصيات لوراثة الأصناف، وإذا حدث هذا، سيكون من الأفضل وضع مرئية محميَّة على الأقل للضابط setter:

```
open class WithInheritance {
   open var isAvailable: Boolean = false
      get() = field
      protected set(value) {
         field = value
      }
}

class WithInheritanceDerived(isAvailable: Boolean) :  WithInheritance() {
   override var isAvailable: Boolean = isAvailable
      get() {
         //do something before returning the value
         return super.isAvailable
      }
      set(value) {
```

```
            //do something else before setting the value
            println("WithInheritanceDerived.isAvailable")
            field = value
        }
        fun doSomething() {
        isAvailable = false
    }
}

val withInheritance = WithInheritanceDerived(true)
withInheritance.doSomething()
println("withInheritance:${withInheritance.isAvailable}")
```

للالـتزام بقواعـد التغليـف، أُتيِحت خاصـيَّة `isAvailable` للاسـتبدال (overrides)، لكن عُيِّن الضـابط setter ليكون خاصًا.

## 4. التهيئة اللاحقة

يجب تهيئـة أي خاصـيَّة ليسـت عدميَّة (non-null) في البـاني، فمـاذا لـو كنت تريـد حقن قيمـة خاصـيَّة عن طريـق حقن التبعيَّة (dependency injection) ولا تريـد التحقـق من العَدم في كل مـرة تصِـل إليها؟ أو ربما عيَّنت قيمــة الخاصــيَّة في إحــدى التوابــع المكشــوفة حســب نوعــك، فتــدعم كــوتلن التهيئــة اللاحقة (Late initialization)، وكل ما عليك القيام به هو استخدام الكلمة المفتاحية `lateinit`:

```
class Container {
    lateinit var delayedInitProperty: DelayedInstance

    fun initProperty(instance: DelayedInstance): Unit {
        this.delayedInitProperty = instance
    }
}

class DelayedInstance (val number:Int)
```

```
...
val container=  Container()
container.initProperty(DelayedInstance(10))
println("with delayed initialization:Number=$
{container.delayedInitProperty.number}")
```

هنالك بعض القيود عند استخدام الخاصيات اللاحقة، أولا، لا يمكن أن يكون نوع الخاصيَّة بدائيًا، وثانيًا، ولا يمكن لخاصيتك استخدام شيفرة جالب getter أو ضابط setter مخصَّصة، وأخيرًا وليس آخرًا، فسينتج عن الوصول إلى خاصيتك قبل تهيئتها رمي الاستثناء kotlin.UninitializedPropertyAccessException.

ليس هنالك سحر عند استخدام lateinit، لنلق نظرة على البايتكود المولَّد لصنف Container:

```
public final class com.programming.kotlin.chapter06.Container {
   public com.programming.kotlin.chapter06.DelayedInstance
delayedInitProperty;
   public final com.programming.kotlin.chapter06.DelayedInstance
getDelayedInitProperty();
      Code:
         0: aload_0
         1: getfield      #11              // Field
delayedInitProperty:Lcom/programming/kotlin/chapter06/DelayedInstance ;
         4: dup
         5: ifnonnull     13
         8: ldc           #12                    // String
delayedInitProperty
         10: invokestatic  #18                    // Method
kotlin/jvm/internal/Intrinsics.throwUninitializedPropertyAccessExcept ion:
(Ljava/lang/String;)V
         13: areturn
```

لقد استبعدت معظم التعليمات البرمجيّة من أجل البساطة، في حين أن الشيفرة البرمجيّة للضابط setter مشابهة للتي أُنشِئَت للصنف Student الذي تحدثنا عنها سابقًا، فإن مجموعة تعليمات الجالب getter مختلفة قليلًا، فالتغيير موجود في السطر 10 حيث سيُرمَى استثناء إذا كان الحقل null.

## 5. الخاصيات المُعمَّمة

تعزِّز كوتلن مفهوم الخاصيات لتشجيع إعادة استخدام الشيفرات البرمجيَّة وجعل مهمة البرمجة أسهل للمطوِّر، فهنالك العديد من مقتطفات البرمجية المتكرِّرة التي يمكنني أنا وأنت كتابتها، ومن الناحية المثاليَّة، يجب أن تملك الوظائف التالية خارج الصندوق:

1. يجب حساب قيمة خاصيّة بتكاسل (lazily) عند أول وصول لها.
2. إخطار المستمعين لقيمة خاصية عند تغييرها، فهل سبق لك وكتبت شيفرة برمجيّة بلغة سي شارب؟ إذا كانت إجابتك نعم، فأنا متأكد من أنك ستتذكر واجهة `InotifyPropertyChange`.
3. استخدام النوع map لتخزين الحقول بدلًا من حقل مادي (materialized field).

حسنًا، إليك خبرًا سارًا! تدعم الخاصيات المُعمَّمة (delegate properties) في كوتلن كل هذا، فنحن نتعامل في كثير من الأحيان مع الأنواع التي تحتاج إلى معرِّف (identifier):

```
interface WithId {
   val id: String
}

data class WithIdImpl(override val id: String) : WithId

class Record(id: String) : WithId by Record.identifier(id) {
   companion object Record {
      fun identifier(identifier: String) = WithIdImpl(identifier)
   }
}
...
val record = Record("111")
println(record.id)
```

لقد رأينا في الفصل الذي يتحدث عن البرمجة كائنيَّة التوجه أنه يمكنك تعميم التوابع (delegate methods)، وينطبق نفس المفهوم على الخاصيات، والصياغة مشابهة أيضًا:

```
val/var<property name>:<Type> by <expression>
```

والتعبير الذي يتبع الكلمة المفتاحية by هو المُعمّم أو المُفوّض الفعلي (actual delegate)، في المثال السابق، قدّمنا خاصيّة للقراءة فقط، إذ لا يعرف المستدعي شيئًا عن WithIdImpl.

لا يحتاج المُعمّم لتطبيق واجهة، فيمكننا تجنّب الوراثة والاعتماد على التكوين فقط، تخيّل أنك تجمع البيانات من جهاز استشعار، وسيحمل كل مقياس منتج على وقت إنشاء الحدث، وسترغب في امتلاك خاصيّة توفّر دعمًا لبصمة الوقت `timestamp` أثناء فرض بعض التحقيقات، ومن أجل البساطة، تركنا جزء التحقق:

```
class TimestampValueDelegate {
   private var timestamp = 0L
      operator fun getValue(ref: Any?, property: KProperty<*>): Long {
         return timestamp;
      }

      operator fun setValue(ref: Any?, property: KProperty<*>, value: Long)
{
         timestamp = value
      }
   }

class Measure {
   var writeTimestamp: Long by TimestampValueDelegate()
}

val measure = Measure()
measure.writeTimestamp = System.currentTimeMillis()
println("Current measure taken at:${measure.writeTimestamp}")
```

قد تجد الشيفرة البرمجيّة السابقة غير عاديّة في البداية، على الأرجح أنت تتساءل ما أول معامليّن لكل تابع في توابع `TimestampValueDelegate`، يُمثّل معامل `ref` النسخة الذي تصل من خلاله إلى الخاصيّة، وفي حالتنا هو نسخة `Measure`، المتغيّر `measure`، ويمثّل معامل الدالة الثانيّة خاصيّة، مثل إعلان `val` أو `var`

مسماة.

يمكنك الحصول على معلومات الخاصيّة باستخدام عامل `::`، في المثال السابق، كل ما عليك فعله هو استخدام `Measure::writeTimestamp`، وإذا أردت دعم خاصيّة القراءة والكتابة فأنت بحاجة إلى توفير جالب get وضابط set، كما فعلنا مع `TimestampValueDelegate`، وإذا كانت خاصيتك للقراءة فقط، أي `val`، فيجب عليك توفير تابع `getValue` فقط، ويجب أن تسبق الكلمة المفتاحية `operator` كلا هاتين الدالتين.

ما السحر الذي يوفِّر كل هذا معًا؟ سننظر إلى البايتكود وسنكشف النقاب عن الآليّة المستخدمة، دعنا نستخدم `javap` مرّةً أخرى للحصول على الشيفرة البرمجيَّة المولَّدة من قبل المصرِّف:

```
public final class com.programming.kotlin.chapter06.Measure {
  public final long getWriteTimestamp();
    Code:
      0: aload_0
      1: getfield      #11                 // Field
writeTimestamp$delegate:Lcom/programming/kotlin/chapter06/TimestampValueDe
legate;
      4: aload_0
      5: getstatic     #15                 // Field  $
$delegatedProperties:[Lkotlin/reflect/KProperty;
      8: iconst_0
      9: aaload
      10: invokevirtual #21                 // Method
com/programming/kotlin/chapter06/TimestampValueDelegate.getValue:(Lja
va/lang/Object;Lkotlin/reflect/KProperty;)J
      13: lreturn
public final void setWriteTimestamp(long);
    Code:
      0: aload_0
      1: getfield      #11                 // Field
writeTimestamp$delegate:Lcom/programming/kotlin/chapter06/TimestampValueDe
legate;
      4: aload_0
```

```
     5: getstatic     #15                 // Field  $
$delegatedProperties:[Lkotlin/reflect/KProperty;
     8: iconst_0
    9: aaload
     10: lload_1
    11: invokevirtual #29                 // Method
com/programming/kotlin/chapter06/TimestampValueDelegate.setValue:(Lja
va/lang/Object;Lkotlin/reflect/KProperty;J)V
     14: return
public com.programming.kotlin.chapter06.Measure();
    Code:
      0: aload_0
      1: invokespecial #35                 // Method
java/lang/Object."<init>":()V
      4: aload_0
      5: new           #17                 // class
com/programming/kotlin/chapter06/TimestampValueDelegate
      8: dup
      9: invokespecial #36                 // Method
com/programming/kotlin/chapter06/TimestampValueDelegate."<init>":()V
     12: putfield      #11                 // Field
writeTimestamp$delegate:Lcom/programming/kotlin/chapter06/TimestampValueDe
legate;
     15: return
static {};
   Code:
      0: iconst_1
      1: anewarray     #51                 // class
kotlin/reflect/KProperty
      4: dup
      5: iconst_0
      6: new           #53                 // class
kotlin/jvm/internal/MutablePropertyReference1Impl
      9: dup
```

```
     10: ldc           #2                  // class
com/programming/kotlin/chapter06/Measure
     12: invokestatic  #59                 // Method  kotlin/jvm/internal/
Reflection.getOrCreateKotlinClass:(Ljava/lang/Class;)Lkotlin/reflect/
KClass;
     15: ldc           #60                 // String writeTimestamp
      17: ldc           #62                 // String
getWriteTimestamp()J
      19: invokespecial #65                 // Method
kotlin/jvm/internal/MutablePropertyReference1Impl."<init>":(Lkotlin/
reflect/KDeclarationContainer;Ljava/lang/String;Ljava/lang/String;)V
     22: invokestatic  #69                 // Method  kotlin/jvm/internal/
Reflection.mutableProperty1:(Lkotlin/jvm/internal
/MutablePropertyReference1;)Lkotlin/reflect/KMutableProperty1;
      25: checkcast     #51                 // class
kotlin/reflect/KProperty
     28: aastore
      29: putstatic     #15                 // Field  $
$delegatedProperties:[Lkotlin/reflect/KProperty;
       32: return
}
```

هذا جزء من البايتكود، سنبدأ من الجزء الأخير من مقتطف الشيفرة البرمجيّة، لاحظ أن المُصرِّف أنشأ بانٍ ساكن (static constructor) لنا وهو المسؤول عن تهيئة حقل ساكن (static field) يسمى `$$delegatedProperties` والذي هو مصفوفة من `Kproperty` (انظر للمدخل `static {}`)، تُنشئ هذه المصفوفة في السطر 1 وفي السطر 29 تُخزَّن في حقل `$$delegatedProperties` الساكن. وابتداءً من السطر 6، يُنشئ نسخة `MutablePropertyReference1Impl` وهو تنفيذ قابل للتغيير (لأننا عرَّفنا حقلنا باستخدام `var`) وخزَّناه كعنصر أول من حقل المصفوفة (انظر للسطر 5).

بالانتقال إلى الشيفرة المولَّدة للباني، يمكننا رؤية أنه يُنشِئ حقلًا من نوع `TimestampValueDelegate` تلقائيًا (انظر للسطر 12)، وضع في اعتبارك أننا نُعمّم الخاصيّة إلى صنف `TimestampValueDelegate`، وبالتالي وجود هذا الحقل.

التوابع getTimestamp و setTimestamp متشابهة تمامًا، لذلك سنتحدّث عن getTimestamp فقط، وهي تستدعي في السطر 10 تابع getValue المكشوف عن طريق TimestampValueDelegate، وتمرِّر مرجع كائن Measure وقيمة Kproperty التي حصل عليها من الحقل الساكن $$delegatedProperties (انظر للسطر 5).

كما ترى، لا يُستخدم السحر عند استخدام الخاصيات المُعمَّمة، لأن المصرّف يُنشئ الشيفرات البرمجيّة المتكرّرة لنا.

هنالك حالات عندما يعرض نوعك الكثير من الحقول وقد لا تكون مهيّئة ومُستخدمة دائمًا، وبالتالي، قد يكون من الأفضل عدم وجود حقلٍ مساعدٍ لكل نوع لتقليل التأثيرات على الذاكرة، وقد ترغب في خزن قيمة كل خاصيّة في map، مما يؤدي إلى عمل إجراء صغير عند البحث، وسيريك المثال التالي كيف يمكنك كتابة شيء مثل هذا:

```
class MapDelegate {
   private val map = mutableMapOf<String, Any?>()
   operator fun <T> getValue(ref: Any?, property: KProperty<*>): T  {
      return map[property.name] as T
   }

   operator fun <T> setValue(ref: Any?, property: KProperty<*>,  value:
T?) {
      map.put(property.name, value)
   }
}

data class SomeData(val char: Char)

class PropsByMap() {
   private val mapDelegate = MapDelegate()
   var p1: Int by mapDelegate

   val p2: SomeData by mapDelegate
```

```
  init {
    mapDelegate.setValue(this, PropsByMap::p2, SomeData('K'))
    mapDelegate.setValue(this, PropsByMap::p1, 0)
  }
}

...

val propsByMap = PropsByMap()
println("Props with map: p1=${propsByMap.p1}")
println("Props with map: p2=${propsByMap.p2}")
propsByMap.p1 = 100
println("Props with map: p1=${propsByMap.p1}")
```

إذا نفَّذت الشيفرة البرمجيّة فسترى 0 و `SomeData(char=K)` و 100.

لحسن الحظ، ليس علينا كتابة تعليمات برمجيّة كما في المثال السابق لأن دعم خاصيات `map-backed` مدمج في كوتلن، وإذا كان لدينا صنف مع خاصيّة للقراءة فقط، فيمكننا كتابة شيء مثل هذا:

```
class Player(val map: Map<String, Any?>) {
  val name: String by map
  val age: Int     by map
  val height: Double by map
}
val player = Player(mapOf("name" to "Alex Jones", "age" to 28,  "height"
to 1.82))
println("Player ${player.name} is ${player.age} ages old and is  $
{player.height} cm tall")
```

إذا كان تصميم الصنف يتطلّب خاصيّة القراءة والكتابة، فسنحتاج إلى استخدام الصنف `Map` قابل للتغيير على عكس المثال السابق الذي استخدمنا فيه تخطيط خريطة map غير قابلة للتغيير.

```
class Player(val map: MutableMap<String, Any?>) {
  var name: String by map
```

```
    var age: Int by map
    var height: Double by map
}
```

تحتوي مكتبة كوتلن على واجهة تساعدك على طلب بصمات التوابع للخاصيات المُعمَّمة، إذا كنت تتعامل مع خاصيَّة للقراءة فقط، كل ما عليك فعله هو الاشتقاق من واجهة `ReadOnlyProperty`، وهنالك واجهة مشابهة لدعم المفوضين لخاصيات القراءة والكتابة، وتسمى واجهة `ReadWriteProperty`، ولست مجبرًا على استخدام هذه الواجهة، فوجودها في الإطار (framework) سيساعدك على الحصول على توقيع التابع الصحيح، بدلًا من أي شيء آخر:

```
data class TrivialProperty(private val const: Int) :
ReadOnlyProperty<Trivial, Int> {
    override fun getValue(thisRef: Trivial, property: KProperty<*>):  Int {
        return const;
    }
}

class Trivial {
    val flag: Int by TrivialProperty(999)
}
...
val trivial = Trivial()
println("Trivial flag is :${trivial.flag}")
```

في حين أن هذه الشيفرة البرمجيَّة لا تقوم بأكثر من إرجاع قيمة، لكن لا يجب أن تستخدمها بهذه الطريقة، فهي تظهر كيف تستخدم الواجهة.

## 6. التهيئة الكسولة

هنالك حالات ترغب فيها بتأخير إنشاء نسخة لكائنك حتى استخدامه الأوَّل، وتُعرف هذه التقنيَّة بالتهيئة الكسولة (lazy initialization / lazy instantiation)، والغرض الرئيسي من التهيئة الكسولة هو تحسين الأداء وتقليل مساحة الذاكرة الخاصة بك، لأن تهيئة نسخة سيتكلف الكثير من العمليات الحسابيَّة وقد لا يستخدمه

البرنامج، لذا سترغب في تأخيره أو تجنب استهلاك دورات وحدة المعالجة المركزيّة (CPU cycles)، تخيَّل أنك تعمل على برنامج لشركة تأمين صحي.

سيكون لديك قائمة من المطالب للعميل، وللحصول عليها، ستحتاج إلى الذهاب إلى قاعدة البيانات وتحميل المعلومة، وهذه العمليَّة مكلفة للغاية، وإذا كان المستخدم لا يهتم بهذه المعلومة، سيكون هذا مضيعة لدورات وحدة المعالجة المركزيَّة (CPU cycles) والذاكرة، فقط عندما يطالبك المستخدم بقائمة المطالب ستذهب وتهيئ مجموعة المطالب.

بالطبع، يجب عليك كتابة شيفرتك البرمجيَّة للتعامل مع التهيئة، لكن قام مطورو كوتلن بهذا العمل من أجلك، قد يبدو التنفيذ الكسول تافهًا في البداية، فبعد كل شيء، عليك التأكد فقط ما إذا كانت القيمة قد عُيّنت بالفعل، أليس كذلك؟ لكن عند التزامن، ستعمل شيفرات البرمجيَّة لتهيئة خاصيتك من قبل خيوط (threads) مختلفة، ويمكنك أن ترى التعقيد المختلف قليلًا هنا، فأنا متأكد من أن أول تطبيق يأتي في ذهن الجميع هو استخدام كتلة التزامن لتحقيق هذا، فعلى الرغم من سهولتها وسرعة كتابتها إلا أنها ستضر بالإنتاجيَّة، فهنالك طرق أخرى لتحسين الشيفرة البرمجيَّة وتجنب القفل.

التزامن ليس للجميع، لذلك، فأني أوصي باستخدام التعليمات المتوفِّرة بدلًا من تنفيذ واحدة خاصة بك، فيوفر كوتلن العديد من التطبيقات التي تناسب جميع احتياجاتك.

للاستفادة من خاصيَّة مُعمَّمة مُهيَّأة بكسل، كل ما عليك فعله هو كتابة `by lazy` وتوفير منطق لإنشاء نسخة، وستتم العناية ببقية الخطوات:

```
class WithLazyProperty {
   val foo: Int by lazy {
      println("Initializing foo")
      2
   }
}
...

val withLazyProperty= WithLazyProperty()
```

```
val total= withLazyProperty.foo + withLazyProperty.foo
println("Lazy property total:$total")
```

إذا شغَّلت الشيفرة البرمجيَّة السابقة، سترى رقم 4 مطبوعًا على الشاشة، لكن سيظهر نص `Initializing` مرَّة واحدة على الرغم من أنك استدعيت الخاصيّة `foo` مرتين.

تأخذ الدالة `lazy` تعبير لامدا - الشيفرة المسؤولة عن إنشاء النسخة - وتُرجع لك نسخة `Lazy<T>` وسيكون تعريف واجهة `Lazy` كالتالي:

```
public interface Lazy<out T> {
   public val value: T
   public fun isInitialized(): Boolean
}
```

يُوفِّر الإطار ثلاثة تعريفات للدالة `lazy`، وهم يغطون جميع الحالات المحتملة، في المثال السابق، استخدمنا هذه:

```
fun <T> lazy(initializer: () -> T): Lazy<T> =
SynchronizedLazyImpl(initializer)
```

من اسم الصنف المُرجع، يمكنك استنتاج ما تفعله، دون النظر إلى التطبيق، فإننا نعلم أن كتلة `initialization` ستعمل داخل كتلة شيفرة المزامنة:

```
private object UNINITIALIZED_VALUE

private class SynchronizedLazyImpl<out T>(initializer: () -> T,  lock:
Any? = null) : Lazy<T>, Serializable {
   private var initializer: (() -> T)? = initializer
   @Volatile private var _value: Any? = UNINITIALIZED_VALUE
   private val lock = lock ?: this

   override val value: T
      get() {
```

```
            val _v1 = _value
            if (_v1 !== UNINITIALIZED_VALUE) {
                @Suppress("UNCHECKED_CAST")
                return _v1 as T
            }

            return synchronized(lock) {
                val _v2 = _value
                if (_v2 !== UNINITIALIZED_VALUE) {
                    @Suppress("UNCHECKED_CAST") (_v2 as T)
                }
                else {
                    val typedValue = initializer!!()
                    _value = typedValue
                    initializer = null
                    typedValue
                }
            }
        }
```

يمسك الصنف دالة لامدا الخاصة بك ويستخدم الحقل lock للحصول على دعم التزامن أثناء تهيئة الحقل value، ولتحسين سرعة الجالب getter، سيضع التنفيذ قيمة افتراضيَّة في الحقل field، وبهذه الطريقة يمكنك اختصار عملية الإرجاع دون الحاجة إلى الحصول على lock، وبالتالي يتحسَّن الأداء؛ يمكنك تزويد النسخة الخاص بك لـ lock إذا أردت المزيد من السيطرة، ويمكننا أخذ المثال السابق واستخدام النسخة الثانية المعاد تعريفها لـ lazy:

```
fun <T> lazy(lock: Any?, initializer: () -> T): Lazy<T> =
SynchronizedLazyImpl(initializer, lock)
class WithLazyPropertyWithLocking{
   val lockingField = Any()
```

```
    val foo: Int by lazy(lockingField, {
      println("Initializing foo");
      2
    })
}
```

تمنحك النسخة الثالثة المعاد تعريفها من دالة lasy المزيد من السيطرة على نوع التنفيذ الكسول المكتوب، وبالتالي، هذه النسخة هي تابع مصنع (factory method):

```
fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T>
```

يمكن أن تأخذ LazyThreadSafteyMode إحدى القيم التالية (الشرح من الشيفرة المصدريَّة):

1. SYNCHRONIZED: يعني هذا استخدام الأقفال (locks) للتأكد من أن خيط (thread) واحد فقط يمكنه تهيئة نسخة [Lazy].
2. PUBLICATION: يعني هذا أنه يمكن استدعاء دالة التهيئة عدة مرات للوصول المتزامن إلى قيمة نسخة [Lazy] غير مهيَّأة، لكن ستُستخدَم القيمة الأولى التي سترجع كقيمة لنسخة [Lazy].
3. NONE: يعني هذا عدم استخدام الأقفال لمزامنة الوصول إلى قيمة نسخة [Lazy]، وإذا تم الوصول إلى النسخة من عدة خيوط، فسيكون سلوكه غير محدَّد. يُستخدَم هذا الوضع عندما يكون من الضروري أن يكون الأداء عاليًا فقط ويضمن لك أنه لن يُهيِّئ نسخة [Lazy] من أكثر من خيط واحد.

إذا استخدمت وضع Synchronized، فسينتهي بك الحال بنفس التنفيذ الذي رأيته سابقًا، وإذا اخترت Publication، فيمكنك استخدام هذا التنفيذ:

```
private class SafePublicationLazyImpl<out T>(initializer: () -> T) :
Lazy<T>, Serializable {
    private var initializer: (() -> T)? = initializer
    @Volatile private var _value: Any? = UNINITIALIZED_VALUE
    // this final field is required to enable safe publication of
constructed instance
    private val final: Any = UNINITIALIZED_VALUE
```

```
  override val value: T
  get() {
    if (_value === UNINITIALIZED_VALUE) {
      val initializerValue = initializer
      // if we see null in initializer here, it means that the value is
already set by another thread
      if (initializerValue != null) {
        val newValue = initializerValue()
        if (valueUpdater.compareAndSet(this, UNINITIALIZED_VALUE,
newValue)) {
          initializer = null
        }
      }
    }
    @Suppress("UNCHECKED_CAST")
    return _value as T
  }
  …
  companion object {
    private val valueUpdater =
java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater(
SafePublicationLazyImpl::class.java,
    Any::class.java,
    "_value")
  }
}
```

لضمان استخدام أوَّل استدعاء من المهيئ، يجب على التنفيذ الاستفادة من `valueUpdater` لتعيين قيمة جديدة تلقائيًا، ومن الداخل، تستخدم تعليمات الجهاز للموازنة والتبديل (swap).

وأخيرًا، إذا اخترت NONE لوضع التزامن، سينتهي الأمر بنسخة `UnsafeLazyImpl` وسيؤدي إلى تحقيق نتائج أفضل لكن يجب استخدامه بشكل مناسب:

```kotlin
internal class UnsafeLazyImpl<out T>(initializer: () -> T) : Lazy<T>,
Serializable {
   private var initializer: (() -> T)? = initializer
   private var _value: Any? = UNINITIALIZED_VALUE

   override val value: T
   get() {
      if (_value === UNINITIALIZED_VALUE) {
         _value = initializer!!()
         initializer = null
      }
      @Suppress("UNCHECKED_CAST")
      return _value as T
   }
```

وفقًا للتوثيق، يجب التأكد من حدوث التهيئة في خيط واحد، وإلا سينتج نسخة مع نفس الحالة (state) وستكون هذه الحالة غير قابلة للتغيير، ويمكنك الحصول على أكثر من خيط يستدعي كتلة التهيئة، وسيُحتفَظ بالقيمة الأخيرة المكتوبة، لكن ستبقى القيمة السابقة ليكنسها كانس المهملات (garbage collector).

توفِّر مكتبة كوتلن القياسيَّة تنفيذًا كسولًا لسيناريوهات عندما تكون القيمة معروفة مسبقًا، فكل ما عليك فعله هو استدعاء `lazyOf(Your_Value)`.

لن تستخدم هذه في العادة، فليس هنالك هدف من تغليف قيمة معروفة لحاوية كسولة (lazy container)، ومع ذلك، قد يكون لديك تسلسلًا هرميًا للصنف الذي يحدّد حقلًا أو تابعًا مثل `Lazy<T>`، ففي مثل هذه الحالة، يمكنك استخدام الباني السابق لإرجاع نسخة `Lazy<T>` مع القيمة المهيَّأة.

## 7. استعمال lateinit مقابل lazy

قد يبدو في البداية أن استعمال `lateinit var` و `by lazy {…}` متشابهين، لكن هنالك اختلافات كبيرة بينهما نلخصها بالنقاط التالية:

1. يمكن استخدام المُعمَّم `lazy {…}` على خاصيات `val` فقط أما `lateinit` فيُستخدَم فقط لخاصيات

.var

2. لا يمكن تصريف خاصيّة `lateinit var` إلى حقل نهائي، وبالتالي لا يمكنك تحقيق الثبات.
3. تملك خاصيّة `lateinit var` حقلًا مساعدًا (backing field) لتخزين القيمة، في حين تُنشئ `lazy {...}` كائنًا مُعمّمًا (delegate object) يعمل كحاوي للقيمة بمجرّد إنشائها ويُوفّر جالب (getter) للخاصيّة، وإذا كنت بحاجة إلى أن يكون الحقل المساعد (backing) موجودًا في الصنف، فيجب عليك استخدام `lateinit`.
4. لا يمكن استخدام خاصيّة `lateinit` للخاصيات القابلة للعَدم (nullable) أو أنواع جافا البدائيّة، فهذا التقييد مفروض من استخدام null لقيم غير مهيئة.
5. خاصيَّة `lateinit var` مرنة أكثر عندما يتعلق الأمر بمكان تهيئتها، فيمكنك إعدادها في أي مكان يكون الكائن مرئيًا منه، وبالنسبة لـ `lazy{}`، فهي تعرِّف المهيئ الوحيد للخاصيَّة، والذي لا يمكن تغييره إلا عن طريق الاستبدال (overriding)، ولذلك سيكون المُهيئ معروفًا مسبقًا، وعلى عكس خاصيَّة `lateinit var`، فعلى سبيل المثال إذا كنت تستخدم حقن التبعيَّة (dependency injection)، فسينتهي بك الأمر بتوفير أمثلة مختلفة من أصناف مشتقَّة.

## 8. المراقبات

ماذا لو أردت معرفة متى تتغيَّر خاصيَّة مُعمَّمة (delegated property)؟ فقد تحتاج إلى التفاعل مع التغيير واستدعاء شيفرات برمجيَّة أخرى.

يأتي كائن `Delegates` مع الباني التالي للسماح لك بذلك:

```
fun <T> observable(initialValue: T, crossinline onChange: (property: KProperty<*>, oldValue: T, newValue: T) -> Unit):
    ReadWriteProperty<Any?, T>
```

سنرى طريقة عمله مع هذا المثال البسيط، ففي كل مرَّة تتغير فيها قيمة الخاصيَّة، سيُستدعى تابع `onValueChanged()` وسنطبع القيمة الجديدة:

```
class WithObservableProp {
    var value: Int by Delegates.observable(0) { p, oldNew, newVal ->
onValueChanged()
}

    private fun onValueChanged() {
       println("value has changed:$value")
    }
}
val onChange = WithObservableProp()
onChange.value = 10
onChange.value = -20
```

هنالك تطبيق آخر لمتغير ظاهر من خارج الصندوق، حيث يسمح لنا برفض قيمة جديدة إذا كان السياق يرفض ذلك:

```
class OnlyPositiveValues {
   var value: Int by Delegates.vetoable(0) { p, oldNew, newVal -> newVal
>= 0 }
}
val positiveVal= OnlyPositiveValues ()
positiveVal.value = 100
println("positiveVal value is ${positiveVal.value}")

positiveVal.value = -100
println("positiveVal value is ${positiveVal.value}")

positiveVal.value = 111
println("positiveVal value is ${positiveVal.value}")
```

إذا قمت بتشغيل الشيفرة البرمجيَّة السابقة، ستجد أنه لن تُقبَل القيمة 100، وبالتالي، فإنه سيطبع 100 مرتين.

## 9. تعميم خاصيَّة لا عدميَّة

إطار كوتلن غني جدًا، فهو يدعم تعميم خاصيَّة لا عدميَّة (non-null)، وكل ما عليك القيام به هو استخدام `Delegates.nonNull` كما في المثال البسيط التالي:

```
class NonNullProp {
   var value: String by Delegates.notNull<String>()
}

val nonNull = NonNullProp()
nonNull.value = "Kotlin rocks"
println("Non null value is: ${nonNull.value}")

//this will not compile
nonNull.value = null
```

إن محاولة الوصول إلى قيمة الخاصيَّة قبل تهيئتها سيؤدي إلى رمي الاستثناء `IllegalStateException`، وعلاوة على ذلك، إذا حاولت تعيين أي قيمة لها، فستحصل على خطأ تصريف.

## 10. الخاصيات أم التوابع؟

تشبه الخاصيات التوابع كثيرًا، فهم يتكونون من الداخل من تابع جالب getter أو ضابط setter كما رأينا بالفعل، ومع ذلك، فإن التوابع والخاصيات تملك أنماط استخدام مختلفة، ويجب عليك اعتبار الخاصيات حقولًا، ففي حين أنهم يشبهون الحقول، فإنَّ صياغة الخاصيَّة تشبه تعاملنا مع الحقل، وتُوفِّر الخاصيات توابع مرنة.

يُمثِّل تابع الصنف إجراءً، بينما تُمثِّل الخاصيَّة بيانات، يجب استخدام الخاصيات كالحقل

وليس كسلوك (behavior) أو عمل (action)، وعندما ترغب في تصميم نوعك وتعريف خاصيَّة أو أكثر، اتبع هذه الإرشادات لتحديد ما إذا كانت مناسبة:

1. تجنَّب وجود شيفرة برمجيَّة معقَّدة في جسم الجالب getter، لأن المُستدعِي يتوقَّع ردًا سريعًا،

وبالتأكيد، لا تتصل بقاعدة بيانات أو إجراء استدعاء `rest` من قاعدة شيفرة البرمجية لجالب الخاصيَّة.

2. لا ينبغي أن يتسبب الحصول على خاصيَّة أية أثار جانبيَّة، تجنب حتى رمي الاستثناءات من شيفرة الجالب (getter).
3. عين الضابط setter الخاص بك على أنه `private` أو `protected` إذا لم ترغب في تغيير القيمة من قبل المستدعي، وهذا يعني أنك تريد الحفاظ على التغليف الخاص بك، تذكر إذا كان نوع الخاصيَّة هو نوع مُرجِع، فسيتمكن المستدعى من تغيير الحالة عن طريق الخاصيات/التوابع العامة المكشوفة.
4. تأكد من تعيين خاصياتك في أي ترتيب ممكن، حتى لو كان ذلك يعني ترك كائنك في حالة غير صالحة مؤقتًا.
5. إذا احتاج الضابط setter لرمي استثناء، فتأكد من الاحتفاظ بقيمة الخاصيَّة السابقة.

هنالك سيناريوهات حيث يجب عليك استخدام تابع بدلًا من خاصيَّة، وعلى الرغم من أننا لا نستطيع تغطية جميع الحالات الممكنة هنا، هذه بعض الحالات التي يجب عليك فيها استخدام التوابع بدلًا من الخاصيات:

1. إذا كانت الشفرة البرمجيَّة أبطأ من عمليَّة إعداد حقل، فاستخدم التابع، فكر في سيناريوهات إعداد قيمة لخاصيَّة تتضمَّن اتصالًا بالشبكة أو حتى الوصول إلى نظام الملفات، ففي هذه الحالات، يجب عليك التأكد من استخدام التوابع بدلًا من الخاصيات.
2. إذا كان استدعاء شيفرة الخاصيَّة يُنتِج نتائج مختلفة في كل مرَّة، فيجب عليك في هذه الحالة استخدام التابع، لنفترض أنك تُرجع الوقت الحالي، فيجب عليك إنشاء تابع له بدلًا من توفير خاصيَّة.
3. إذا أردت تحويل نوع إلى نوع آخر مختلف، فيجب عليك استخدام التابع، ومثال ذلك دالة `toString()`، فأي إعلان تجد فيه النمط `to***` يجب أن يكون تابع بدلًا من خاصيَّة.
4. إذا كان الهدف نسخ الحالة الداخليَّة لكائنك، فلا يجب استخدام الخاصيَّة بل التابع، ومثال ذلك تابع `clone` المُعرَّف من قبل صنف كائن جافا.

## 11. خلاصة الفصل

لن تحتاج الآن إلى كتابة أو توليد جالبات getters وضابطات setters لحقولك، فتعتمد التقنيات التقليديَّة للتغليف بشكل حصري على التوابع المنفصلة، لكن تسمح لك الخاصيات الآن بالوصول إلى حالة كائن باستخدام صياغة تشبه الحقل مع الاحتفاظ بالتغليف، ولقد تعلَّمت الآن ما هي الخاصيات وكيف تُستخدَم.

ستتعرف في الفصل القادم على كيفية عمل خاصيات لغة كوتلن الجديدة لإزالة استثناء مؤشر العدم، وعلاوةً على ذلك، سترى كيف تتكامل شيفرات العدم في جافا مع هذه الخاصيات.

الفصل السابع:

# أمان القيم الفارغة، والانعكاس، والتوصيفات



إن استثناء مؤشر القيمة المعدومة أو الفارغة (null pointer) مألوف لأي مطوِّر جافا مهما كانت المدة الـتي قضاها في التطوير، ويحدث هذا الاستثناء بسبب فشل معالجة مراجع هذه القيمة معالجـة صحيحة، وكـان "تجنب هذه الأخطاء" موضوعًا للعديد من الأفكار المختلفة في لغات برمجة عديدة، وفي هـذا الفصـل، سـنراجع نهج كـوتلن في تأمين القيم الفارغة أو المعدومة، ولقد قال توني هاور (Tony Hoare)، مخترع مؤشر العَدم، عنـد تحدثـه في مؤتمر Qcon المنظم من قبل موقع تطوير التدوين InfoQ:

> **أسـميها خطـأ المليـار دولار الخاصـة بي، ولقـد اخترعتهـا عـام 1965، وفي ذلـك الوقت، كنت أُصمّم أول نظام نـوع شامل للمراجع في لغـة كائنيّة التوجّـه (لغـة ALGOL W)، إذ كـان هـدفي التأكـد من اسـتخدام جميـع المراجـع بشـكل آمن عن طريـق إجـراء فحص تلقـائي بواسـطة المصـرّف، ولم اسـتطع مقـاوم إغـراء وضـع مرجـع عَـدم، لأنـه كـان من السـهل تنفيـذه، ولقـد أدى هـذا إلى عـدد لا يحصـى من الأخطـاء وثغـرات وانهيـارات في النظـام والـذي تسـبّب في خسـارة مليار دولار في السنوات الأربعين الماضية.**

هنالك العديد من الطرائق لحل هذا الخطأ، ففي لغة السي، كان من الشائع أن تتسبَّب شـيفرة برمجيـة تشـير إلى مؤشر عَدمي إلى إنهيار البرنامج، ولقـد طـوَّر جافـا هـذا من خلال الاسـتثناء `NullPointerException` حـتى لا يتسـبب في انهيـار JVM، ويمكن التعامـل معـه عن طريـق كتلـة `try/catch`، وكـان العبء تـذكر الإمسـاك بـه على المبرمج.

قدَّمت لغات Groovy و #C ميزات مصـمَّمة للسـماح للمصـرَّف بإمسـاك شـيفرة عدميَّة (nullable code) محتملة وحماية المطـوَّر، وأمـا في لغـات مثـل سـكالا وهاسـكل وغيرهـا من لغـات البرمجـة الإجرائيَّة، فهنالـك النـوع Maybe والنوع Option المكتوبان بنمط التصميم monad المخصصان لذلك.

تمتلـك كـوتلن طريقـة لتحقيـق الأمن من العَـدم داخـل نظـام الأنـواع، فلا يُمثِّل أمـان القيم العدميـة أو الفارغـة باستخدام حاوية عَدم مصممة بالنمط monad (أي monadic null container)، ولا عن طريق إجبار المبرمج بإمساك الاستثناء، وبدلًا من ذلك، تمت إضافة الدعم مباشرةً إلى نظام النوع والمصرِّف.

وستكون الميزات التي سنتحدث عنها في هذا الفصل مألوفة لدى بعض القراء، فهنالك ميزات مماثلـة في لغـات مثل groovy و #C بالفعل، فسنغطي:

- الأنواع القابلة للإنعدام (Nullable types) والأنواع غير القابلة للإنعدام (Non-nullable types)
- عوامل الأمان من القيم المعدومة (Null safe operators)
- فحص الانعكاس (Reflection) والشيفرة البرمجيّة وقت التشغيل
- التوصيفات (Annotations)

## 1. الأنواع القابلة للإنعدام

إن نظام أنواع كوتلن متطوّر بما فيه الكفاية بحيث يمكنه تتبُّع الفرق بين الأنواع التي تقبل قيم عدمية أو فارغة من تلك التي لا تقبلها، فعندما نحدّد متغيّرًا في كوتلن، كما نفعل في العادة، لا يمكننا تعيين قيمة عَدم `null` له، فعلى سبيل المثال، لن تُصرَّف هذه الشيفرة البرمجيّة:

```
val name: String = null // لا تصرَّف الشيفرة
```

ولن يُصرَّف إسناد قيمة عَدميّة `null` إلى متغيّر من النوع `var` أيضًا:

```
var name: String = "harry"
name = null // لا تصرَّف الشيفرة
```

ولإبلاغ مصرِّف كوتلن بأننا نسمح لمتغيّر أن يحتوي على قيمة `null`، يجب أن نضيف لاحقة `?` للنوع:

```
val name: String? = null
var name: String? = "harry"
name = null
```

سيُصرَّف الآن المقتطف البرمجي السابق.

وبنفس الطريقة، يمكننا إرجاع أنواع عدميّة وغير عدميَّة من دالة واستخدامها كمعاملات دالة وهكذا:

```
fun name1(): String = ...
fun name2(): String? = ...
```

لا يمكن للدالة `name1` إرجاع مرجع عدمي (null reference)، ويمكن للدالة `name2` إرجاع ذلك أو عدم إرجاعه، وإذا كان علينا كتابة شيفرة برمجيَّة تستخدم نتيجة `name1` فسنضمن عندها أنه لن يُرمى استثناء مؤشر العدم (null pointer exception)، ولكن إذا حاولنا كتابة تعليمات برمجيَّة تصل إلى نتيجة `name2` فلا يمكن ضمان ذلك، وبالتالي فلن يقبل المصرِّف الشيفرة البرمجيَّة دون معالجة إضافيَّة.

## 2. التحويل الذكي بين الأنواع

لقد رأينا الآن كيف نعلن عن أنواع يمكن أن تقبل القيم العدمية أو الفارغة (nullable-type)، لكن السؤال هنا: كيف نستخدم مثل هذه الأنواع ونتعامل معها؟ الخيار الأول هو بالاعتماد على التحويل الذكي (smart casts) بين الأنواع الذي عرَّفناه باختصار في الفصل الثاني، أساسيات كوتلن، والذي هو ميزة من مميزات كوتلن إذ يتبع المصرِّف الشروط داخل تعليمة if الشرطية، فما دمنا نتحقَّق من أن المتغير ليس عَدمي، فسيسمح لنا المصرِّف بالوصول إلى المتغيِّر كما لو صرِّح عنه على أنَّه نوع غير عَدمي:

```
fun getName(): String? = ...
val name = getName()
if (name != null) {
    println(name.length)
}
```

لاحظ أنَّنا قادرون على استدعاء دالة الطول على قيمة `name` داخل تعليمة `if`، وهذا لأن المصرِّف تأكَّد من أنَّه لا يمكننا تنفيذ ما داخل تلك الكتلة حتى يكون المتغيِّر `name` قيمة غير عَدميَّة.

| تنبيه | لا يعمل التحويل الذكي للأنواع العدمية (null smart cast) إلا عندما يكون المتغير إما عضوًا من النوع `val` دون حقل مساعد (backing field) أو `val` محلي أو `var` محلي لا يتغير بين التحقُّق منه واستخدامه، وخلاف ذلك، قد يكون المتغير غير عَدمي عند التحقُّق منه ومن ثم سيتغيَّر إلى عَدمي قبل استخدامه، وسيؤدي هذا إلى رمي استثناء آنذاك. عمومًا، سيفرض المصرِّف هذا القيد. |
| --- | --- |

## 3. الوصول الآمن للقيم الفارغة

إن التحويل الذكي للأنواع هي ميِّزة جيِّدة للغاية، وتقدم طريقة سهلة القراءة للتفرّع عند التعامل مع القيم العدميَّة، ومع ذلك، عندما تكون لدينا عمليات متسلسلة، فقد تنتج عن كل خطوة قيمة فارغة أو عدميَّة، وستُصبح الشيفرة البرمجيَّة صعبة القراءة وتحتاج إلى فحص وتمحيص دقيق.

اطلع على المقتطف التالي:

```
class Person(name: String, val address: Address?)
class Address(name: String, postcode: String, val city: City?)
class City(name: String, val country: Country?)
class Country(val name: String)

fun getCountryName(person: Person?): String? {
   var countryName: String? = null
   if (person != null) {
      val address = person.address
      if (address != null) {
         val city = address.city
         if (city != null) {
            val country = city.country
            if (country != null) {
               countryName = country.name
            }
         }
      }
   }
   return countryName
}
```

انظر إلى مستويات التحقق من القيم الفارغة (if-not-null) المتداخلة المطلوبة، ويمكن تخيل المزيد من المستويات المتداخلة المطلوبة في بعض السيناريوهات، فهل يمكننا فعل أفضل من ذلك؟

نستطيع فعل ذلك باستخدام كوتلن، فبديل التحويل الذكي هو استخدام عامل الوصول الآمن للقيم الفارغة، ويشبه هذا صياغة النقطة العاديَّة للدوال والخاصيات، ولكن باستخدام `?`، فعند استخدام هذا العامل، سيُدخِل المُصرِّف تحقيق `null` لنا للتأكد من أننا لا نصل إلى قيمة فارغة `null` عن طريق الخطأ، ويمكننا كتابة المثال السابق على النحو التالي:

```
fun getCountryNameSafe(person: Person?): String? {
      return person?.address?.city?.country?.name
}
```

الفرق مدهش، وإذا فحصنا البايتكود المولّد لهذه الدالة، فيمكننا أن نرى أن المصرِّف أضاف تحقيقات القيم الفارغة:

```
public static final java.lang.String getCountryNameSafe(Person);
   Code:
      0: aload_0
      1: dup
      2: ifnull        32
      5: invokevirtual #15    // Method Person.getAddress:()LAddress;
      8: dup
      9: ifnull        32
      12: invokevirtual #21     // Method Address.getCity:()LCity;
      15: dup
      16: ifnull        32
      19: invokevirtual #27   // Method City.getCountry:()LCountry;
      22: dup
      23: ifnull        32
      26: invokevirtual #33        // Method  Country.getName:()Ljava/lang/
String;
      29: goto          34
      32: pop
      33: aconst_null
```

```
      34: areturn
```

التعليمات المهمة هنا هي 2، 9، 16 و23، والتي تعرض إجراء المصرِّف للتحقق من القيم الفارغة وإذا كانت هنالك قيمة فارغة، فسيقفز إلى التعليمة 32 قبل إضافة `null` إلى المكدِّس الذي سيُرجَع.

## أ. عامل تحويل النوع

قد نقرّر في بعض الأحيان الاستغناء عن تحقيقات المصرِّف وإجبار تحويل نوع عَدمي إلى نوع غير عَدمي، وهذا مفيد في بعض الحالات كالتعامل مع شيفرات برمجيَّة مكتوبة بلغة جافا، والتي نعرف أنها لا تكون عَدميَّة أبدًا، وسنحتاج إلى استخدام متغيِّر مع دالة لا تقبل سوى قيم غير عَدميَّة أو فارغة، وللقيام بذلك، يمكننا استخدام العامل `!!` كما في هذا المثال:

```
val nullableName: String? = "george"
val name: String = nullableName!!
```

في المثال السابق، صرّحنا عن المتغيِّر `name` على أنه نوع غير عَدمي، واستخدمنا `!!` لنفرض على المتغيِّر `nullableName` -الذي يقبل قيمًا فارغة- عدم احتوائه على تلك القيم الفارغة، ويمكننا فعل ذلك لأي تعبير يُرجع قيمةً فارغةً، فعلى سبيل المثال، في المقتطف التالي، تُرجع الدالة قيمة فارغة، لكن أجبرنا المصرِّف على السماح لنا بالتعامل معها على أنَّها قيمة غير فارغة:

```
fun nullableAddress(): Address? = ...
val postcode: String = nullableAddress()!!.postcode
```

يمكننا أن نرى أن العامل `!!` يجعل الشيفرة البرمجيَّة غير آمنة، وإذا تحققنا من بايتكود:

```
public static final void forceFunction();
  Code:
     0: invokestatic  #62                 // Method nullableAddress:
()LAddress;
     3: dup
     4: ifnonnull     10
     7: invokestatic  #67    // Method
```

```
kotlin/jvm/internal/Intrinsics.throwNpe:()V
     10: invokevirtual #70      // Method Address.getPostcode:()Ljava/lang/
String;
     13: astore_0
     14: return
```

في التعليمة 7، سيُرمى استثناء مؤشر العدم إذا حوى المتغيّر postcode القيمة null.

## 4. عامل ألفيس

واحدة من السيناريوهات الأكثر شيوعًا هو عندما يكون لدينا نوع عَدمي ونرغب في استخدام قيمته إذا كان غير عَدمي ونستخدم قيمة افتراضيَّة أخرى خلاف ذلك، فعلى سبيل المثال، في جافا، سنكتب في العادة شيفرة برمجيَّة مشابهة لهذه:

```
String postcode = null
if (address == null) {
   postcode = "No Postcode"
}
else {
   if (address.getPostcode() == null) {
      postcode = "No Postcode"
   }
   else {
      postcode = address.getPostcode()
   }
}
```

تقدم لنا كوتلن بديلًا يدعى بالعامل ?: والذي يدعى بعامل ألفيس، وإذا نظرت إلى العامل بشكل جانبي فسيبدو كتسريحة شعر ألفيس (وتدعى هذه التسريحة بتسريحة بومبادور، لكن قد يكون من الأفضل أن يكون للعامل اسمًا آخر). إن استخدام هذا العامل مشابه لاستخدام if الثلاثيَّة في جافا.

يمكن وضع هذا المعامل بين تعبير عَدمي والتعبير المراد استخدامه إذا كانت قيمة التعبير العَدمي (nullable expression) هي null، ولذلك فإن الاستخدام العام يشبه ما يلي:

```
val nullableName: String? = ...
val name: String = nullableName ?: "default_name"
```

التعبير في الجانب الأيمن، لذلك يمكن وضع أي شيء هناك لتقييم القيمة، مثل تعبير when أو استدعاء دالة، ويمكن سَلسلة العمليات أيضًا.

ومن الطرائق الشائعة الأخرى هو استخدام عامل الوصول الآمن للعَدم لسَلسلة التعابير التي تقبل قيمة فارغة معًا، وقبل استخدام عامل ألفيس Elvis ثم استخدامه لإرجاع قيمة افتراضية عند وجود قيمة فارغة أو معدومة لتلك التعابير:

```
val nullableAddress: Address? = null
val postcode: String = nullableAddress?.postcode ?: "default_postcode"
```

## 5. التحويل الآمن بين الأنواع

لقد تعرفنا في الفصل الثاني، أساسيات كوتلن، إلى عامل as لتحويل نوع متغيِّر آليًا إلى نوع آخر، وإذا أردنا التحويل إلى نوع آخر ولكن بشكل آمن أو استعمال القيمة null في حالة فشلت عملية التحويل، فيمكننا في هذه الحالة استخدام عامل التحويل الآمن as?.

في المثال التالي، سنحوِّل نوع متغيِّر نعرف أنَّه سلسلة نصية في حين أن المصرِّف لا يعرف ذلك لأننا أعلنا على أنَّه Any وذلك باستعمال عامل التحويل الآمن:

```
val location: Any = "London"
val safeString: String? = location as? String
val safeInt: Int? = location as? Int
```

## 6. النوع Optional

تحدثنا في الأجزاء السابقة عن كيفية اتخاذ إجراءات للسلامة من القيم الفارغة أي null، لكن تلك الطرائق ليست الوحيدة، فلقد وفَّرت لغات مثل هاسكل بديلًا لسنوات عديدة، ففي حالة هاسكل، هنالك النوع Maybe، وفي لغة سكالا هنالك شيء مشابه يدعى النوع Option، وفي الإصدار 11 المستقر من جافا (وقت ترجمة الكتاب ومراجعته)، كان هنالك النوع Optional.

جميع هذه الأنواع (Maybe و Option و Optional) لها وظيفة واحدة، وهي استخدامها كنوع لمعرفة ما إذا كانت دالة أو تعبير سيرجع قيمة أو لا أي يرجع قيمة أو تكون القيمة المعادة فارغة.

في البرمجة الإجرائيَّة، أنواع البيانات التي إما أن تعيد قيمة أو لا هي الأنواع الجبريَّة، إذ تمثِّل الأولى قيمة والثانيَّة الافتقار إلى قيمة، وتسمى في هاسكل باسم Just و Nothing، وتسمى في سكالا باسم Some و None، ويُستخدَم نوع وحيد في جافا.

بالنسبة لبقيَّة هذا القسم، سنركِّز على النوع Optional في جافا، وتذكَّر أن هذا لا يعمل إلا على الإصدار 11 من جافا حاليًا (تأكد من توثيق جافا الرسمي دومًا).

### أ. إنشاء قيمة من النوع Optional وإرجاعها

يمكننا تغليف قيمة في النوع Optional عن طريق استدعاء التابع الساكن of ببساطة:

```
val optionalName: Optional<String> = Optional.of("william")
```

إذا اردنا إنشاء متغيِّر من النوع Optional لقيمة فارغة، فيمكننا استخدام التابع الساكن empty:

```
val empty: Optional<String> = Optional.empty()
```

إذا اردنا إنشاء متغيِّر من النوع Optional قد يحوي القيمة null أو لا، فيمكننا استخدام ofNullable، وفي العادة، هنالك نسخة واحدة فارغة (empty) موجودة، لأنها غير قابلة للتغيير ولا تملك حالة.

لذا، إذا كان لدينا دالة ترجع null، فسنعرفها لتعيد ?String، وعند استخدام Optional، فسنعرفها لترجع Optional<String>:

```
fun lookupAddress(postcode: String): String?
fun lookupAddress(postcode: String): Optional<String>
```

يمثِّل هذان المقتطفان نفس الشيء وذلك لدالة lookupAddress يمكن أن لا ترجع قيمة عند تمرير قيمة دخلٍ إليها.

### ب. استخدام النوع Optional

يشبه النوع `Optional` لعوامل العَدم (null operators) في كوتلن من ناحية العمليات المسموحة، وعند استخدام `Optional`، سنحتاج إلى استخراج القيمة منه، ولفعل هذا، يمكننا استخدام `get` أو `orElse` ويسترد الأول القيمة أو يرمي استثناءً ويشبه هذا عامل التحويل بين الأنواع، وأما الثاني فيقبل معاملًا يستخدمه كقيمة افتراضية إذا لم يمثِّل `Optional` أي قيمة أي كان `null`, إليك المثال التالي:

```
fun lookupAddress(postcode: String): Optional<String> = ...
val address = lookupAddress("AB1 1BC").orElse("1600 Pennsylvania  Avenue")
```

في المثال السابق، إذا لم تمتلك الدالة `lookupAddress` عنوانًا للرمز البريدي الموجود، فستُستخدَم القيمة الافتراضيّة، ويشبه هذا العامل `?:`.

يدعم النوع `Optional` العمليتين `map` و `flatMap` لتحويل القيمة الحاوية، فتقبل عملية `map` دالة تُرجع قيمة جديدة من النوع `Optional` مع نتيجة الدالة، وإذا كان `Optional` فارغًا، فلن تُستدعَى الدالة. وتعمل عملية `flatMap` بشكل مشابه، لكنها تعمل على تسطيح الأنواع `Optional` المتداخلة المعادة بواسطة الدالة. لنتخيَّل دالة أخرى تُرجع `Optional<Int>`:

```
fun lookupHousePrice(address: String): Optional<Int> = …
```

يمكننا الآن ربط هذه مع `lookupAddress` لإيجاد سعر المنزل إذا كان `address` موجودًا، وإذا كان سعر المنزل غير موجود (لنفترض أنه ليس في قاعدة البيانات)، فلن تُرجع `flatMap` `Optional<Optional<Int>>` بل سيُسطِّح النوع المتداخل ليكون نوع الإرجاع هو `Optional<Int>`:

```
val price = lookupAddress("AB11BC").flatMap(::lookupHousePrice).orElse(0)
```

بما أنَّ لدينا `Optional<Int>` بعد `flatMap`، فيمكننا استخدام `orElse` لإرجاع عدد صحيح افتراضي.

## 7. الانعكاس

الانعكاس (Reflection) هو اسم عملية فحص الشيفرة البرمجية وقت التشغيل بدلًا من وقت التصريف،

ويمكن استخدامها لإنشاء نسخ من أصناف، والبحث عن دوال، واستدعاؤها، وتفقّد التوصيفات (annotations)، إيجاد الحقول واكتشاف المعاملات والأنواع المُعمَّمة (generics)، وكل هذا دون معرفة التفاصيل وقت التصريف.

فعلى سبيل المثال، قد نرغب في الاحتفاظ بأنواع في قاعدة البيانات، لكننا لا نعرف، أو لا نريد أن نعرف مقدمًا أي الأنواع التي سيُحتفَظ بها، فيمكن هنا استخدام الانعكاس للبحث عن حقول كل نوع، وإنشاء شيفرة SQL البرمجية المناسبة لكل نوع.

مثال آخر، إذا كان لدينا نظام ملحقات (plugin system) في شيفرتنا المصدرية، ونرغب في وقت التشغيل إنشاء نُسخٍ للملحق بناءً على خاصيات تكوين أو خاصيات النظام، فيمكننا استخدام الانعكاس لاستنساخ أصناف بناءً على الاسم المؤهل بالكامل (fully qualified name) الذي مررناه.

بالنسبة لبقيَّة هذا الفصل، سنغطي مختلفة أصناف الانعكاس والدوال الذي أتاحته كوتلن في حزمة الانعكاس.

**تنبيه**

لا تُعدُّ أصناف الانعكاس في كوتلن جزءًا من مكتبة كوتلن القياسية kotlin-stdlib بل هي جزءٌ من تبعيَّة إضافيَّة (additional dependency) تسمى kotlin-reflect، وهذا لإبقاء حجم الحزم منخفض لمستخدمي أندرويد والمنصات ذات الذاكرة المقيَّدة.

## أ. النوع Kclass

نوع KClass هو النوع المركزي المستخدم في موضوع الانعكاس في كوتلن، فكل نوع له نسخة KClass وقت التشغيل تحتوي على تفاصيلٍ عن دوال وخاصيات وتوصيفات وما إلى ذلك، وللحصول على نسخة KClass لأي نوع، يمكننا استخدام الصياغة ::class على نسخة ذلك النوع:

```
val name = "George"
val kclass: KClass<String> = name::class
```

لاحظ أن نسخة KClass موسومة بالنوع الذي تمثله، ويمكننا أيضًا الحصول على KClass للنوع باستخدام نفس الصياغة على النوع نفسه:

```
val kclass2: KClass<String> = String::class
```

توجد نسخة KClass واحد لكل محمِّل صنف (class loader) وذلك لأي نوع، لذلك، إن استدعاء ::class على أي نسخة في نفس محمِّل الصنف سيُرجع نفس نسخة KClass المعادة عند استدعائه على أية نُسخٍ أخرى مشتقة من النوع الذي يمثه ذلك الصنف، أو من النوع نفسه:

```
val kclass1: KClass<String> = "harry"::class
val kclass2: KClass<String> = "victoria"::class
val kclass3: KClass<String> = String::class
```

في المثال السابق، تُشير جميع المتغيرات KClass الثلاثة إلى نفس النسخة.

وبصرف النظر عن استرداد مقبض (handle) لـ KClass عبر النُسخ والأنواع، يمكننا أيضًا الحصول على واحد من الاسم المؤهل بالكامل للصنف، ولفعل هذا، سنحتاج إلى جلب مرجع إلى الواجهة البرمجية (API) لانعكاس جافا المعادل لـ KClass، والذي يسمى Class ببساطة، ومن ثم نصل إلى خاصيَّة كوتلن كما يوضح المثال التالي:

```
val kclass = Class.forName("com.packt.MyClass").kotlin
```

إن التابع الساكن Class.forName هو واجهة الانعكاس البرمجية في جافا لاسترداد مقبض لنسخة Class، وفي الحقيقة، استوحيت أسماء دوال عديدة في KClass من التوابع الموجودة في Class، لكن حُدِّثت لدعم ميزات كوتلن المتقدمة.

### ب. التهيئة باستخدام الانعكاس

كما ذكرنا سابقًا، أحد أكثر استخدامات الانعكاس هو إنشاء نُسخٍ من أنواع دون معرفة هذه الأنواع وقت التصريف، وأبسط طريقة لفعل هذا هو استخدام الدالة createInstance على مرجع KClass:

```
class PositiveInteger(value: Int = 0)
fun createInteger(kclass: KClass<PositiveInteger>): PositiveInteger {
     return kclass.createInstance()
}
```

كما ترى، تستخدم الدالة `createInteger` الخاصة بنا معامل `KClass` لإنشاء نسخة `PositiveInteger` جديدة، فلا يُستخدَم هذا المثال المفتعل عند معرفة النوع مسبقًا، لكن هدف الانعكاس هو لتلك الأوقات التي لا تفعل ذلك.

إن عيب `createInstance` هو أنه سيعمل فقط مع الأصناف بدون معاملات، أو عندما تكون جميع المعاملات اختياريَّة، يعتبر المعامل اختياريًا إذا وفّرت له قيمة افتراضيَّة.

لنفترض حالة استخدام نموذجيَّة لهذا النوع من التهيئة، في تطبيق معالجة البيانات، قد يكون لدينا خطوة استيراد، والتي تستورد أو تبتلع (ingests) البيانات من ملفات CSV إلى قاعدة البيانات الخاصة بنا؛ سيعمل تطبيقنا على ابتلاع البيانات من مصادر عديدة، ونريد أن نكون قادرين على إضافة مدخلات جديدة وقت التشغيل دون الحاجة إلى إعادة بناء الشيفرة البرمجيَّة الرئيسيَّة.

سنبدأ بتعريف ملف `config.` والذي سيحتوي على قائمة من البالعات (ingesters)، ويُشار إلى كل بالع عن طريق الاسم المؤهل بالكامل (FQN)، أي اسم الصنف مسبوق باسم الحزمة)، وسيبدو مثل التالي:

```
ingesters.props
ingesters=com.packt.ingester.AmazonIngester,com.packt.ingester.Goo
gleIngester
```

سنرغب في بعض أصناف Bootstrap تحميل ملف الخاصيَّة هذا، وتقسيم السلاسل النصيَّة للحصول على أسماء البالعات، ومن ثم نُهيئها باستخدام الانعكاس، وبمجرَّد الحصول على مرجع إلى كل البالعات، يمكن أن نستدعي كل واحدة بدورها، ويتطلّب هذا تنفيذ واجهة مشتركة مع دالة نقطة الدخول (entry point function):

```
interface Ingester {
   fun ingest(): Unit
}

val props = Properties()
props.load(Files.newInputStream(Paths.get("/some/path/ingesters.pr ops")))
val classNames = (props.getProperty("ingesters") ?: "").split(',')
```

```
val ingesters = classNames.map {
   Class.forName(it).kotlin.createInstance() as Ingester
}

ingesters.forEach { it.ingest() }
```

لاحظ أننا عرّفنا في البداية الواجهة `Ingester` والتي سيُوسّعها كل تطبيق بالع، عندما نهيئ كل بالع بشكل انعكاسي (reflectively)، فإننا مطالبون بتحويل الأنواع إلى أنواعها التي عرَّفناها بها صراحةً، وسترمي هذه العملية استثناء إذا كان الصنف في ملف الإعدادات `config` ليس من نوع `Ingester`، ومن الواضح أن المصرِّف غير قادر على تأكيد ذلك لنا بناءً على سلسلة نصية وحدها.

الفائدة من هذا النهج هو حاجتنا إلى إضافة بالع إضافي، ولنقل مثل كتابة تنفيذ من أجل `com.packt.ingester.FacebookIngester` ومن ثم عدم الحاجة للمس شيفرة التطبيق الأساسية، ويمكننا نشر بالع الجديد في ملف JAR منفصل، وإضافةً ملف jar إلى مسار الصنف وتحديث الملف `config` لتضمين الاسم المهيأ كاملًا FQN الجديد.

هذا الأسلوب شائع للأنظمة التي تعتمد على ملحقات (Plugins) حيث لا يمكن لمطوري النظام الأساسي معرفة التنفيذات (implementations) التي ستُكتب عند استخدام مكتباتهم.

تذكر أنَّ `createInstance` لا تدعم استعمال المعاملات، وقد لا يبدو من المفيد جدًا إنشاء نُسخٍ بشكل انعكاسي دون معاملات، لكن في حالات مثل المثال السابق، لا يمكننا أن ندعم جميع أشكال البانيات التي سيرغب بانو الملحقات استخدامها، لذا قد نقيِّدهم دون أي معاملات، ونفرض عليها إنشاء مفوَّضين (delegates) بالشكل المطلوب.

## 8. البانيات

قد نرغب أحيانًا في فحص البانيات (constructors) المتاحة على نوعٍ ما، وسنحتاج ربما إلى إنشاء نوع يمتلك بانٍ يتطلَّب قيم أو قد نرغب بتحديد أي الحقول مطلوبة لإنشاء نسخة من نوعٍ وقتَ التشغيل، أو، بالمثل، قد نرغب بمعرفة ما إذا كان بالإمكان إنشاء صنف من المعاملات التي نمتلكها آنذاك.

يمكننا إرجاع قائمة بجميع البانيات المتوفرة لنوع معيّن باستخدام الخاصيَّة constructors المتاحة في النوع KClass، وترجِع هذه الخاصيَّة قائمة من نُسخ KFunction المنعكسة، لأن البانيات هي دوال بحد ذاتها لكن فقط دوال معرَّفة بطريقة خاصة:

```
fun <T : Any> printConstructors(kclass: KClass<T>) {
 kclass.constructors.forEach {
   println(it.parameters)
  }
}
```

يكرر المثال السابق فوق كل باني، ويطبع المعاملات التي يقبلها، وعلى سبيل المثال، انظر إلى الصنف المعرَّف التالي:

```
class Kingdom(name: String, ruler: String, peaceful: Boolean) {
   constructor(name: String, ruler: String) : this(name, ruler,  false)
}
```

إذا استدعينا printConstructors لهذا النوع:

```
fun main(args: Array<String>) {
   printConstructors(Kingdom::class)
}
```

فستكون المخرجات مثل تالي:

```
[parameter #0 name of fun <init>(kotlin.String, kotlin.String):  Kingdom,
parameter #1 ruler of fun <init>(kotlin.String, kotlin.String):  Kingdom]
[parameter #0 name of fun <init>(kotlin.String, kotlin.String,
kotlin.Boolean): Kingdom,
parameter #1 ruler of fun <init>(kotlin.String, kotlin.String,
kotlin.Boolean): Kingdom,
parameter #2 peaceful of fun <init>(kotlin.String, kotlin.String,
kotlin.Boolean): Kingdom]
```

يمكننا استدعاء مرجع إلى بانٍ باستخدام الدالتين call و callBy المتاحتين، فتوجد نسختين منهما، تقبل

الأولى ببساطة قائمة المعاملات أي varargs ويفترض أن يكونوا مرتبين بالشكل الذي صُرِّح عنه مسبقًا في الباني، ويقبل الإصدار الثاني خريطة map من المعاملات وتستخدم أسماء المعامل لمطابقتهم:

```
fun createKingdom(name: String, ruler: String, peaceful: Boolean): 
Kingdom {
   val constructor = Kingdom::class.constructors.find {
      it.parameters.size == 3
   } ?: throw RuntimeException("No compatible constructor")
   return constructor.call(name, ruler, peaceful)
}
```

في الشيفرة المصدريَّة السابقة، استخدمنا المتغيِّر الأول الذي يمرِّر المعاملات بالترتيب.

عند إنشاء أمثلة بشكل انعكاسي، يجب علينا أن نضمن أن الأنواع متوافقة، وإذا كان على السبيل المثال، المعامل الأول المتوقع هو java.lang.String ومرَّرنا java.math.BigDecimal فسترمي آلة جافا الافتراضية JVM استثناءً من النوع java.lang.IllegalArgumentException.

## أ. الاستنساخ باستخدام callBy

الدالة callby، التي تستخدم خريطة map، مفيدة للغاية إذا رغبنا في بناء الوسائط (arguments) الملائمة بشكل انعكاسي بنفسها، ولبناء هذه الخريطة، يمكننا استخدام المعلومات حول المعاملات التي تقدمها البانيات لنا عن طريق المعاملات المسماة للخاصية (property named parameters).

ترجع هذه الخاصية عدة نسخ من KParameter، واحدة لكل معامل في الباني، ويمكن استخدام معاملات نسخ الانعكاس هذه لتحديد اسم ونوع المعامل وسواء كانت varargs أو inline أو Optional.

دعنا نشرح هذا بإنشاء نسخة Plugin، وهو نوع نرغب بإنشائه يقبل إما اتصال JDBC، أو نسخة من خاصيات (properties instance)، أو نظام الملفات FileSystem. وفي وقت التصريف، لن نعرف أي معامل سيحتاج إليه الباني من بين هذين المعاملين، وسنستخدم الانعكاس لنعرف ذلك.

سنُعرِّف واجهة Plugin، بالإضافة إلى كتابة تنفيذ وهمي يسمى OraclePlugin:

```
interface Plugin {
   fun configure(): Unit
}

class OraclePlugin(conn: Connection) {
   fun configure(): Unit = ... // run queries on the connection
}
```

لاحظ أن `OraclePlugin` يقبل النسخة `Connection`، والهدف الأساسي من هذا المثال هو الشيفرة البرمجيّة للانعكاس التي ستُنشِئ هذه الإضافات:

```
fun createPlugin(className: String): Plugin {
 val kclass = Class.forName(className).kotlin
 assert(kclass.constructors.size == 1, { "Only supply plugins with a
single constructor" })
 val constructor = kclass.constructors.first()

assert(constructor.parameters.size == 1, { "Only supply plugins with one
parameter" })
val parameter: KParameter = constructor.parameters.first()

val map = when (parameter.type.jvmErasure) {
   java.sql.Connection::class -> {
      val conn = DriverManager.getConnection("some_jdbc_connection_url")
      mapOf(parameter to conn)
   }
   java.util.Properties::class -> {
        val props = Properties()
        mapOf(parameter to props)
      }
      java.nio.file.FileSystem::class -> {
        val fs = FileSystems.getDefault()
        mapOf(parameter to fs)
```

```
      }
      else -> throw RuntimeException("Unsupported type")
    }

    return constructor.callBy(map) as Plugin
  }
```

أولًا، يمكننا استخدام `Class.forName` كما في السابق للحصول على مرجع لنسخة من `KClass` من أجل الوسيط `className`، ومن ثم، بالاعتماد على ذلك، جلبنا أول بانٍ وأوّل معامل له، وفي هذه الحالة بالذات، نتوقَّع من تنفيذات الملحق أن تمتلك بانٍ واحدٍ فقط مع معامل وحيد، وسنضيف تأكيدات لهذا الغرض.

وبعد ذلك، نفحص النوع الذي يُمثله `KParameter`، وبالاعتماد على النوع، سنبني خريطة map تحتوي على قيمة لأحد الأنواع المدعومة: `Connection` و `Properties` و `FileSystem`.

وأخيرًا، نُمرِّر تلك الخريطة `map` إلى الدالة `callBy` من أجل استنساخ نسخة وتهيئتها، مع تحويل الأنواع للحصول على النوع المطلوب.

## 9. الكائنات والكائنات المرافقة

يمكننا حتى الحصول على مرجع إلى كائنات (objects) أو كائنات مُرافِقة (companion objects) عن طريق الانعكاس، وعلى سبيل المثال، خذ هذه التعريفات لصنف ولكائن مرافق:

```
class Aircraft(name: String, manufacturer: String, capacity: Int) {
  companion object {
    fun boeing(name: String, capacity: Int) = Aircraft(name, "Boeing",
capacity)
  }
}
```

وبالنظر إلى هذا، يمكننا استرداد مرجع إلى الكائن المرافق (companion object) باستخدام الخاصيَّة `companionObject` المعرَّفة في النوع `KClass`:

```
val kclass = Aircraft::class
val companionKClass = kclass.companionObject
```

من الآن فصاعدًا، لدينا نسخة KClass أخرى، ويُنمذِج (modeling) هذا الدوال (functions) والأعضاء (members) للكائن المرافق.

في الواقع، يمكننا باستخدام خاصيَّة companionObjectInstance الحصول على مقبض (handle) لنسخة الكائن المرافق، وبعد ذلك، يمكننا استدعاء الدوال أو الوصول إلى الخاصيات مباشرةً إذا قمنا بالتحويل إلى النوع المناسب:

```
val kclass = Aircraft::class
val companion = kclass.companionObjectInstance as  Aircraft.Companion
companion.boeing("747", 999)
```

لاحظ أن نوع الكائن المرافق الذي حولنا إليه هو Aircraft.Companion ولقد كان كائنًا مرافقًا بدون اسم.

وبطريقة مماثلة، إذا كان لدينا KClass يمثِّل كائن منفرد (object singleton)، فيمكننا استخدام الخاصيَّة objectInstance لاسترداد النسخة الفعلية:

```
object PizzaOven {
   fun cook(name: String): Pizza = Pizza(name)
}

val kclass = PizzaOven::class
val oven: PizzaOven = kclass.objectInstance as PizzaOven
```

كما ترى، فإن المتغيِّر الأخير oven هو نسخة للكائن PizzaOven.

## 10. خاصيات KClass المفيدة

يصف KClass صنفًا محددًا بشكل كامل بما في ذلك نوع المعاملات والأصناف الأعلى (superclass) والدوال والبانيات والتوصيفات. دعنا نعرّف الصنف التالي:

```
class Sandwich<F1, F2>()
```



يمكننا الآن فحص الصنف KClass له ومعرفة أنواع المعاملات التي يصرِّح عنها، وذلك باستخدام الخاصيَّة typeParameters المتاحة في نسخة KClass:

```
val types = Sandwich::class.typeParameters
```

ومن هنا، يمكننا الحصول على تسمية (Label) لمعامل النوع، والحدود العليا إذا عُرِّف إحداها (أو Any خلا ذلك):

```
types.forEach {
     println("Type ${it.name} has upper bound ${it.upperBounds}")
}
```

في حالة Sandwich، ستكون المخرجات كالتالي:

```
Type F1 has upper bound [kotlin.Any?]
Type F2 has upper bound [kotlin.Any?]
```

بعد ذلك، دعنا نعرض الصنف الأعلى لنوع معيَّن، سنحتاج أولًا إلى نوع يملك العديد من الآباء:

```
class ManyParents : Serializable, Closeable, java.lang.AutoCloseable
```

وبعد ذلك، في نسخة KClass من هذا الصنف، يمكننا الوصول إلى الخاصيَّة superclasses للحصول على قائمة الأصناف العليا والواجهات، لكن ليس الصنف الفعلي نفسه:

```
val superclasses = ManyParents::class.superclasses
```

إذا توجب عليها طباعة القائمة السابقة، فسنرى المخرجات التالية:

```
class java.io.Serializable
class java.io.Closeable
class java.lang.AutoCloseable
```

هذا ما توقعناه بالطبع، لكن ماذا عن الصنف Any؟ ألا تتوسع جميع الأصناف منه؟ فلماذا لم نجدها ضمن القائمة؟ هذا لأن الأصناف العليا تشمل فقط الآباء المباشرين، ولعرضها، نحتاج إلى استعمال الخاصيَّة

:allSuperclasses

```
val allSuperclasses = ManyParents::class.allSuperclasses
```

وسنحصل على الخرج التالي:

```
class java.io.Serializable
class kotlin.Any
class java.io.Closeable
class java.lang.AutoCloseable
```

## 11. الدوال والخاصيات المنعكسة

لا يتوقف الانعكاس على الأصناف والكائنات، فيمكن الوصول إلى معظم نظام كوتلن، ويشمل هذا الدوال والخاصيات؛ ولنبدأ بصنف يحتوي على بعض دوالٍ أعضاءٍ ودالة مُوسّعة للنوع Double وبعض من خاصياته:

```
class Rocket() {
   var lat: Double = 0
   var long: Double = 0

   fun explode() {
      println("Boom")
   }

   fun setCourse(lat: Double, long: Double) {
      require(lat.isValid())
      require(long.isValid())
      this.lat = lat
      this.long = long
   }

   fun Double.isValid() = Math.abs(this) <= 180
}
```

تُستخدَم دالة مُوسّعة للتأكد من أنّ معامل Double هو خط العرض أو الطول الصحيح كلما استدعينا setCourse.

تشبه الدالة التالية دالة طباعة البانيات السابقة، وهي تطبع أسماء كل دالة مُعرَّفة في هذا الصنف، وتُستخدم الخاصيَّة memberFunctions في KClass للحصول على مرجع لكل دالة في الصنف، وتُمثَّل الدوال في الواجهة البرمجية للانعكاس عن طريق نُسخ KFunction:

```
fun <T : Any> printFunctions(kclass: KClass<T>) {
     kclass.functions.forEach {
          println(it.name)
     }
}
```

إذا استدعينا هذه الدالة، فسنحصل على المخرجات التالية:

```
explode
setCourse
equals
hashCode
toString
```

كما هو متوقَّع، يحتوي هذا الخرج على دوال الأعضاء التي عرَّفناها في Any: النوع الأعلى النهائي (ultimate supertype) لجميع الأصناف، ولاحظ مع ذلك، أن الدالة المُوسّعة ()Double.isValid لا تظهر في القائمة، وسنحتاج من أجل الحصول على مرجع KFunction من أجل دالة مُوسّعة إلى استخدام خاصيَّة أخرى تسمى memberExtensionFunctions.

هنالك خاصيَّة ثالثة، الدوال المسماة (named functions) ببساطة، والتي ترجع كل من الدوال المُوسّعة وغير المُوسّعة في نفس القائمة، وهي تشبه الجمع بين مخرجات الخاصيتين السابقتين.

لدى نُسَخ KFunction العديد من دوال والخاصيات المفيدة، وتُستخدم لاكتشاف تفاصيل مثل إذا ما كانت الدالة inline، أو infix، أو operator، بالإضافة إلى النوع الذي ترجعه، وأنواع المعاملات وغيرها من التفاصيل.

وعندما يتعلّق الأمر بالخاصيات، هنالك خاصيات مماثلة تسمى memberProperties و memberExtensionProperties والتي تُستخدَم بنفس الطريقة كما في الدوال. في الواجهة البرمجية للانعكاس، تُمثّل الخاصيات من خلال نُسخ Kproperty. دعنا نستخدم memberProperties للعثور على الخاصيات التي عرّفناها في الصنف Rocket:

```
fun <T : Any> printProperties(kclass: KClass<T>) {
     kclass.memberProperties.forEach {
          println(it.name)
     }
}
```

ستكون مخرجات هذه الدالة عند استدعائها: lat, long كما هو متوقع.

## أ. استدعاء دالة بشكل انعكاسي

تكمن الفائدة الحقيقيّة في الوصول الانعكاسي (reflective access) إلى الدوال في القدرة على استدعائها، فيُعرِّف الصنف KFunction دالةً تسمى call تقبل قائمة من المعاملات (vararg)، وتستخدمها لاستدعاء دالة على نسخة النوع الذي صُرّحت تلك الدالة به.

وبالنظر إلى أن نُسخ KFunction نفسها ليست مرتبطة بأي نسخة بعينها، سنحتاج إلى توفير المتلقي الذي يجب على الدالة أن تُستدعَى عليه، ويحدّد دائمًا بالمعامل الأول المُمرَّر إلى call.

باستخدام مثال Rocket السابق، سنستدعي function ديناميكيًا، دون توفير مرجع إليها في وقت التصريف (compile time):

```
val function = kclass.functions.find { it.name == "explode" }
val rocket = Rocket()
function?.call(rocket)
```

لاحظ أنّنا نبحث عن جميع الدوال التي تحتوي على اسم explode، والدالة explode لا تُصرِّح في الواقع عن أي معاملات بنفسها، ولذلك فإن المعامل الوحيد للدالة call هو النسخة المراد استخدامها كمستقبل أو متلقي

للدالة function، وفي هذه الحالة، وهو rocket.

### ب. التصريح وعدمه

في هذه المرحلة، يجب الإشارة إلى الفرق بين الدوال والخاصيات المُصرَّح عنها (declared) وغير المُصرَّح عنها (undeclared)، وكل من الخاصيات التي يمكن استخدامها لجلب دوال أعضاء (member functions)، وخاصيات أعضاء (member properties)، وبانيات وما إلى ذلك والتي تختلف إن كان مُصرَّح عنها أم لا.

تشمل المتغايرات الغير مُصرَّح عنها (undeclared variants)، التي قمنا بتغطيتها لحد الآن، بما في ذلك الدوال والخاصيات المُصرَّح عنها في النوع المشار إليه من قبل KClass، وكذلك للأصناف الأم والواجهات.

تشمل المتغايرات المُصرَّح عنها (declared variants)، والتي تسمى declaredMemberExtensionFunctions، و declaredMemberFunctions وغيرها، على الدوال والخاصيات المعرَّفة في النوع نفسه، وأي دالة أو خاصيَّة معرَّفة في الأصناف الآباء (parent classes) أو الواجهات التي لا تعيدها هاتان الدالتان.

## 12. التوصيفات

تسمح التوصيفات (Annotations) للمطورين بإضافة معنى إضافي إلى الأصناف، والواجهات، والمعاملات، وغيرها وذلك وقت التصريف، وهي شكل من أشكال البرمجة الوصفيَّة (meta-programming). يمكن أن يستخدم المصرِّف تلك التوصيفات أو تستخدمها الشيفرة مباشرةً عن طريق الانعكاس وقت التشغيل، وبالاعتماد على قيمة التوصيف، يمكن أن يتغيَّر معنى البرنامج أو البيانات.

التوصيفات موجودة في جافا كما في كوتلن، وبالتالي فإن التوصيفات الأكثر شيوعًا هي تلك المتوفِّرة كجزء من مكتبات كوتلن أو مكتبة جافا القياسيَّة. يمكن أن تكون على دراية بالفعل ببعض التوصيفات مثل @SuppressWarnings و @tailrec.

لتعريف توصيف خاص بك، أضف ببساطة الكلمة المفتاحية annotation قبل الصنف كالتالي:

```
annotation class Foo
```



يمكن استخدام هذا التوصيف في الأصناف، والدوال، والمعاملات، وما إلى ذلك، وفي واقع يمكن استخدامها في أي مكان تقريبًا، كما هو موضَّح في الجدول التالي:

| الهدف | مثال |
| --- | --- |
| صنف | `@Foo class MyClass` |
| واجهة | `@Foo interface MyInterface` |
| كائن | `@Foo object MyObject` |
| معامل | `fun bar(@Foo param: Int): Int = param` |
| دالة | `@Foo fun foo(): Int = 0` |
| كنية لنوع | `@Foo typealias MYC = MyClass` |
| خاصيّة | `class PropertyClass { @Foo var name: String? = null}` |
| بان | `class Bar @Foo constructor(name: String)` |
| تعبير | `val str = @Foo "hello foo"` |
| قيم إرجاع | `fun expressionAnnotation(): Int { return (@Foo 123)}` |
| دالة مُجرَّدة | `@Foo { it.size > 0 }` |

لاحظ أن التوصيفات تبدأ بالعلامة @ عند استخدامها، ومع ذلك، قبل استخدامها، يجب تحديد الأهداف المسموح بها باستخدام توصيف ثانوي يسمى `@Target`. فعلى سبيل المثال، لتوصيف بانٍ فقط لصنف، يمكننا تعريفه على النحو التالي:

```
@Target(AnnotationTarget.CONSTRUCTOR)
annotation class Woo
```

يمكننا تحديد العديد من الأهداف التي نريدها لأي توصيف معيَّن.

هنالك العديد من التوصيفات الثانويَّة متاحة للاستخدام، وهي مشروحة في هذا الجدول:

| اسم التوصيف | الاستخدام |
|---|---|
| `@Retention` | يحدد كيفيّة تخزين التوصيف في ملفات الصنف الناتجة.<br>الخيارات هي:<br>• `Source`: يُحذف التوصيف وقت التشغيل.<br>• `Binary`: يتضمن التوصيف في ملفات الصنف، لكنه غير مرئي من خلال الانعكاس.<br>• `Runtime`: يُخزّن التوصيف في ملفات الصنف وهو مرئي من خلال الانعكاس. |
| `@Repeatable` | تسمح إذا كانت موجودة بتضمين التوصيف أكثر من مرة في أي هدف معيَّن. |
| `@MustBeDocumented` | إذا كانت موجودة، فستسمح بتضمين التوصيف عند إنشاء توثيق عبر Dokka. |

## أ. معاملات التوصيف

يمكن للتوصيفات أن تحتوي على معاملات كما رأينا ذلك بالفعل مع `@AnnotationTarget`. ويمكن للتوصيفات المخصَّصة تحديد بانياتها الخاصة مع أي معامل ترغب به، ولفعل ذلك، سنعرِّف بانٍ كما نفعل للصنف العادي عن طريق سرد المعاملات بعد اسم الصنف، وإليك المثال التالي:

```
annotation class Ipsum(val text: String)
```

| تنبيه | يجب التصريح عن معاملات التوصيف على أنَّها `val` دائمًا. |
|---|---|

بعد ذلك، عند استخدام توصيف مثل هذا، نمرِّر القيمة المطلوبة بكل بساطة:

```
@Ipsum("Lorem") class Zoo
```

تقتصر أنواع المعاملات على مجموعة ضيقة من الأنواع هي: `Int` - `Double` - `Long` - `Float` - `enum` - `KClass` - `String` - `Boolean`، وتوصيفات أخرى نفسها، بالإضافة إلى مصفوفة من تلك الأنواع المسموح بها، ولذلك على سبيل المثال، يمكننا الحصول على مصفوفة من النوع `String` أو مصفوفة من التوصيفات.

## 13. التوصيفات القياسيَّة

تتضمَّن مكتبة كوتلن القياسيَّة العديد من التوصيفات التي تُأثِّر على مخرجات المُصرِّف، ولقد رأينا بعضها بالفعل والبعض الآخر سنتحدث عنه في هذا الجزء للمرة الأولى.

### أ. @JvmName

بسبب الماسح أو المزيل (erasure) في JVM، فإنَّه من المستحيل التصريح عن دالتين بنفس الاسم ونفس البصمة المزالة (erased signature)، فعلى سبيل المثال، التصريحات التالية في جافا ستتسبَّب في خطأ عند التصريف:

```
public void foo(list: List<String>)
public void foo(list: List<Int>)
```

| تنبيه | يرجع السبب في الإزالة (Erasure) إلى حقيقة أن آلة جافا الافتراضية JVM لا تحتفظ بأنواع المعاملات، وهذا يعني، أن متغيرات من نوع `List<String>` أو `List<Int>` يصرَّفان كليهما إلى النوع `List<Any>`. |
|---|---|

الحل الأكثر استخدامًا لهذه المشكلة هي تسمية التوابع تسميةً مختلفةً، وهذا غير مرغوب فيه في بعض الأحيان. ففي كوتلن، يمكننا الاحتفاظ بنفس الأسماء طالما نحن نوفِّر أسماءً بديلةً عند تصريفها، ولفعل ذلك، نضع توصيفًا للدوال باستخدام `@JvmName` مع البديل المراد توفيره كما هو موضَّح في الأمثلة التاليَّة:

```
@JvmName("filterStrings")
```

```
fun filter(list: List<String>): Unit

@JvmName("filterInts")
fun filter(list: List<Int>): Unit
```

سيُستخدَم الاسم المعطى للتوصيف وقت التصريف، ويمكننا أن نرى هذا عن طريق فحص البايتكود المُولَّد:

```
public static final void filterStrings(java.util.List<java.lang.String>);
     Code:
          [ ... ]
public static final void filterInts(java.util.List<java.lang.Integer>);
     Code:
          [ ... ]
```

عند استخدام هذه الدوال من كوتلن، نستمر في استخدام الأسماء الأصلية، وسيكون التوصيف `@JvmName` غير مرئي لمستخدمي كوتلن، وسيراه المُصرِّف ويتصرَّف وفقًا له، ولكن عند استدعاء هذه الدوال من جافا، فيجب أن نستخدم الاسم البديل المُعطَى.

### ب. @JvmStatic

يُخبِر التوصيف `@JvmStatic` المصرِّف رغبتك بأن تمتلك الدالة أو الخاصيَّة الموصوفة به تابع جافا ساكنٍ بانٍ مُولَّدٍ في الخرج المُصرِّف (compiled output)، ويمكن تطبيق هذا التوصيف على الكائنات أو الكائنات المرافقة.

بشكل افتراضي، يُصرَّف كائنٌ أو كائنٌ مرافقٌ إلى صنف يحتوي على نسخة واحد، وتُخزَّن هذه النسخة في حقل ثابت اسمه `INSTANCE`، وللوصول إلى الدوال في هذه الكائنات في جافا، يجب عليك أولًا استبيان (resolve) تلك النسخة المفردة (singleton) مثل:

```
HasStaticFuncs.INSTANCE.foo();
```

ومع ذلك، سيؤدي التوصيف إلى أن تكون الدالة تابعًا ساكنًا (static method) بدلًا من تابع نسخة (instance method)، وبالتالي نتمكن من استدعائها بشكل مباشر على النوع:

```
HasStaticFuncs.foo();
```

### ت. @Throws

بما أنَّ جميع الاعتراضات أو الاستثناءات في كوتلن هي استثناءات غير متحقَّق منها (unchecked exceptions)، فلا داعي لإضافة قائمة بالاستثناءات المحتملة إلى بصمات التابع كما في جافا. ومع ذلك، نرغب في إبلاغ مستخدمي جافا بأن واجهة برمجة التطبيقات لدينا ترمي استثناءات في حالات معيَّنة، ويمكننا القيام بذلك باستخدام التوصيف `@Throws`، والذي يُستخدَم لإرشاد المصرِّف لإنشاء عبارات رمي (throw) على التوابع المولَّدة.

فعلى سبيل المثال، لنعرِّف صنفًا بسيطًا في كوتلن يحتوي على دالة يمكنها رمي استثناء:

```
class File(val path: String) {
     fun exists(): Boolean {
          if (!Paths.get(path).toFile().exists())
          throw FileNotFoundException("$path does not exist")
          return true
     }
}
```

يمكن استدعاء هذا مباشرةً من جافا بالطريقة التالية:

```
public void throwsExample() {
     boolean exists = new File("somefile.txt").exists();
     System.out.println("File exists");
}
```

لاحظ أنَّ بصمة التابع لا تتضمَّن الاستثناءات التي يمكن أن ترمى، على الرغم من أنَّ هذه الشيفرة البرمجية قابلة للتصريف. ومع ذلك، ينبغي لنا اتخاذ قرار بشأن إبلاغ مستخدمي جافا بأنَّ الدالة `exists()` ترمي استثناءً، ويمكننا إضافة هذا إلى بصمة التابع في ملف الصنف المُصرَّف:

```
class File(val path: String) {
```

```
    @Throws(FileNotFoundException::class)
    fun exists(): Boolean {
        if (!Paths.get(path).toFile().exists())
    throw FileNotFoundException("$path does not exist")
        return true
    }
}
```

كما ترى، لقد أضفنا التوصيف `@Throws`، ويقبل هذا التوصيف وسيطًا وهو أصناف الاستثناءات التي نرغب في تضمينها في بصمة التابع، فلا يمكن تصريف مثال جافا السابق الآن ويجب علينا تحديثه للتعامل مع الاستثناء بالشكل التالي:

```
public void throwsExample() throws FileNotFoundException {
    boolean exists = new File("somefile.txt").exists();
    System.out.println("File exists");
}
```

أخيرًا، يمكننا ملاحظة الفرق في ملف البايتكود لأن مصرِّف كوتلن يضيف استثناءات لبصمة التابع بالتأكيد.

تحتوي الدالة `exists` الأولى على ترويسة البايتكود التالية:

```
public final boolean exists();
    descriptor: ()Z
    flags: ACC_PUBLIC, ACC_FINAL
```

أما الثانيّة، فتحتوي على ترويسة البايتكود التالية:

```
public final boolean exists() throws
    descriptor: ()Z
    flags: ACC_PUBLIC, ACC_FINAL
```

### ث. @JvmOverloads

غطينا `@JvmOverloads` مسبقًا، لكنّنا سنتوسع بالحديث عنه هنا أكثر. عند إعطاء دالة تملك معاملات

افتراضيَّة، سيجعل `@JvmOverloads` المصرِّف ينشئ توابع متعدِّدة ومحمَّلة تحميلًا زائـدًا (overloaded) لكـل معامل افتراضي.

اطلع على هذه الدالة على سبيل المثال:

```
fun foo(name: String = "Harry", location: String = "Cardiff"):
```

سيولِّد المصرِّف ثلاثة توابع في صنف الملف المُصرَّف، التابع الأول يسـتخدم المعـاملات الافتراضـيَّة عنـد عـدم وجود معاملات،والثاني يستخدم القيمة الافتراضية للمعامل `location` عند عدم تمريره مـع تمريـر قيمـة للمعامـل الآخر، والثالث يستعمل القيم المعطاة له دون القيم الافتراضـية. ويمكننـا التأكـد من ذلـك عن طريـق عـرض بـايتكود المولَّد:

```
public static void foo$default(com.packt.SomeClass, java.lang.String,
java.lang.String, int, java.lang.Object);
descriptor:
(Lcom/packt/SomeClass;Ljava/lang/String;Ljava/lang/String;ILjava/lang/
Object;)V
flags: ACC_PUBLIC, ACC_STATIC, ACC_BRIDGE, ACC_SYNTHETIC
     Code:
          [ ... ]
public final void foo(java.lang.String, java.lang.String);
     descriptor: (Ljava/lang/String;Ljava/lang/String;)V
     flags: ACC_PUBLIC, ACC_FINAL
          Code:
               [ ... ]
public void foo(java.lang.String);
     descriptor: (Ljava/lang/String;)V
     flags: ACC_PUBLIC
          Code:
               [ ... ]
public void foo();
     Code:
          0: aload_0
```

```
1: aconst_null
2: aconst_null
3: iconst_3
4: aconst_null
5: invokestatic #41
8: return
```

تؤكد هذه الشيفرة أن الإصدارات الثلاثة من التابع foo قد تم تعريفها. التنفيذات موجودة للجميع باستثناء الأخيرة فلقد حذفت هنا لأنها تُمثِّل بمئات الأسطر أي للاختصار، ويمكن رؤية الفكرة العامة من تنفيذ الدالة final. كل نسخة من foo تفوِّض (delegate) افتراضيًا التنفيذ إلى دالة رابعة تدعى foo$default مع استخدام null لقيمة المعاملات حيثما كان ذلك مناسبًا، وهذه الأخيرة تتحقق من كل معامل وتستبدله بقيمته الافتراضية إن كانت قيمته null.

## 14. اكتشاف التوصيف وقت التشغيل

التوصيفات المخصصة تكون مفيدة فقط إذا كان من الممكن اكتشافها واستخدامها. التوصيفات القياسيَّة موجودة لتوجيه المصرِّف والاستفادة منه، ولكن شاع استخدام التوصيفات المخصصة كبيانات وصفيَّة وقت التشغيل.

للعثور على التوصيفات الموجودة في صنف، أو دالة، أو بانٍ آخر، يمكننا استخدام الخاصيَّة annotation الموجودة في KClass، و KFunction، و KParameter، و KProperty والتي تُرجع مجموعةً تحتوي على نسخة لكل توصيف معرَّف.

على سبيل المثال، لننشئ توصيفًا يسمى Description والذي يقبل معاملًا واحدًا من نوع String، وتُستخدَم هذه السلسلة النصيَّة لإضافة وصف إلى الصنف، والذي يمكن استخدامه لإنشاء توثيق في خدمة ويب (web service):

```
annotation class Description(val summary: String)
```

ثم سنستخدم هذا لوصف الصنف:

```
@Description("This class creates Executor instances") class Executors
```

الآن، في وقت التشغيل، يمكننا البحث عن هذا التوصيف واستخدامه:

```
val desc = Executors::class.annotations.first() as Description
val summary = desc.summary
```

من الواضح أنَّنا في هذا المثال لم نضف أي شيفرة متعلقة بالبرمجة دفاعيَّة (Defensive programming) التي يتطلبها الواقع، مثل الشيفرة التي من شأنها أن تحقق من أن الصنف يحتوي على التوصيفات، أو أنه يمثّل نوع التوصيف الصحيح.

| تنبيه | تذكر أنه لكي تكون التوصيفات المخصَّصة قابلة للاكتشاف بواسطة الانعكاس، يجب أن يملك التوصيف القيمة `@Retention` القيمة `RUNTIME`. |
|---|---|

## 15. خلاصة الفصل

رأينا خلال هذا الفصل كيف تأمن كوتلن من شر القيم الفارغة أو المعدومة `null`، وكيف يمكن استخدام الانعكاس لفحص الشيفرة البرمجيَّة وقت التشغيل، ولقد تعرفنا أيضًا على عدة توصيفات في كوتلن وشرحنا تأثيرها على المصرِّف.

في الفصل القادم، سنبدأ مناقشة نظام الأنواع المتقدم في كوتلن وكيف يمكننا كتابة شيفرة تحوي أنواعًا مُعمَّمةً (Generics).

الفصل الثامن:

# التعميم والأنواع المُعمَّمَة



الأنواع المُعمَّمَة (Generics أو generic programming) هي تقنيَّة يمكن بواسطتها كتابة دالة عمومية لاستعمالها مع مختلف الأنواع أي عدم اقتصار استخدامها على نوع محدَّد. الأنواع المُعمَّمَة أو Generics هو المصطلح المستخدم في جافا وكوتلن، لكن تُستخدَم أسماء أخرى مثل التعدُّدية الشكلية (parametric polymorphism) والقوالب (templates)، في لغات أخرى بمميِّزات مماثلة.

سنغطي في هذا الفصل:

- إعداد معاملات النوع (Type parameterization)
- حدود النوع (Type bounds) وحدود نوع العودي (recursive type bounds)
- اللاتباين (Invariance) والتغاير (covariance) والتغاير المضاد (contravariance)[11]
- أنواع البيانات الجبريَّة (Algebraic data types)

## 1. دوال ذات معاملات غير محدَّدة النوع

فكِّر في دالة تسمى `random()` والتي إن مرَّرت لها عناصر مختلفة، ترجع عنصرًا واحدًا عشوائيًا منها، فلا نحتاج حينئذٍ إلى معرفة أنواع هذه العناصر عندما نكتب تلك الدالة، كما أننا لن نستعمل العناصر بأنفسنا، ونحتاج فقط إلى اختيار واحد منها لإرجاعه، وعندما نستخدم النوع بهذه الطريقة (تجريد النوع)، نستعمل المصطلح «معامل النوع» أو «معامل مُعمَّم النوع» (type parameter)، لذا سيكون للدالة `random` الخاصة بنا معامل نوع واحد، وهو نوع العناصر التي سنختار أحدها.

إذا أردنا كتابة دالة مُعمَّمَة النوع، مثل دالة `random()` التي ذكرناها للتو، فربما قد نبدأ بشيء مثل هذا:

```
fun random(one: Any, two: Any, three: Any): Any
```

سيعمل هذا في أي نسخة نختارهh، ومع ذلك، بغض النظر عن الأنواع التي نختار تمريرها إليها كمعاملات، سيكون النوع الذي يتم إرجاعه هو `Any`، ومن ثم نضطر إلى التحويل والعودة إلى النوع الأصلي، وهذا الأمر يقودنا إلى الوقوع في الخطأ، ناهيك عن شناعته.

11 هذه الترجمات ابتكرتها اعتمادًا على علم الرياضيات وشرح ويكيبيديا ولم أجد أي ترجمة عربية لها وقد لا تجدها إلا في هذا الكتاب وربما في كتب لاحقة إن جرى اعتمادها على نطاق أوسع.

يمكننا أن نفعل أفضل من هذا باستخدام معامل النوع الذي من شأنه تثبيت الأنواع والسماح للمصرِّف باستنتاج قيمة الإرجاع استنتاجًا صحيحًا، ولتعريف دالة مع معامل نوعٍ، نستخدم صياغة الأقواس الزاويَّة `<...>`، وذلك عن طريق إعطاء معامل النوع اسمًا قبل اسم الدالة:

```
fun <T> random(one: T, two: T, three: T): T
```

في هذا التعريف، قمنا بتعريف معامل نوع واحد باسم T والذي استخدمناه للمعاملات الثلاثة بالإضافة إلى نوع الإرجاع، وبذلك نخبر المصرِّف أن النوع T الذي سيُحدَّد في البداية هو نوع الإرجاع أيضًا. ويسمح هذا للمصرِّف باستنتاج نوع الإرجاع بشكل صحيح.

لاستدعاء هذه الدالة، لا نحتاج إلى القيام بأي شيء سوى تمرير القيم، واحترام العلاقة بينها:

```
val randomGreeting: String = random("hello", "willkommen", "bonjour")
```

يمكنك ملاحظة التصريح عن أن المتغيِّر `randomGreeting` هو من نوع `String`، وهذا فقط لمطابقتها مع قيمة الإرجاع والتي هي سلسلة نصيَّة أيضًا، ولكن في الواقع يُستنتج هذا مثل المعتاد.

ماذا نعني باحترام العلاقة بين الأنواع؟ هذا معناه ببساطة أنه عند استخدام نوع معامل معيَّن، يجب أن يشير إلى نفس النوع، وفي مثالنا، كان لدينا معامل نوع واحد وعيَّن إلى `String` عند استدعاء الدالة، ولذلك، تُرجع الدالة سلسلة نصيَّة `String` أيضًا.

بالطبع، في هذا المثال، كان من الممكن أن نمرِّر أية قيم نريدها، وستُصرَّف الشيفرة مع ذلك، وسيُستنتَج أدنى نوعٍ عالٍ شائعٍ (lowest common supertype) فعلى سبيل المثال، هذه الشيفرة البرمجيّة تصرَّف تصريفًا صحيحًا:

```
val any: Any = random("a", 1, false)
```

هذا لأن جميع الأنواع لها نوع Any الأعلى، وبالتالي يمكن للمصرِّف استنتاج أن T يجب أن تكون Any وستسوفي القيود بذلك.

ومع ذلك، بالنسبة للأمثلة الأخرى، قد لا يعمل هذا، على سبيل المثال، إن قبول قائمة من النوع T وإضافة نوع

T آخر إليها، ففي هذه الحالة، يجب أن تكون القائمة والعنصر المراد إضافته إليها متوافقان.

سنرى مثالًا ملموسًا على ذلك في قسم تعدّد الأشكال المقيّد عندما نتحدث عن الحدود العليا.

يمكن للدوال أن تحتوي بالطبع على أكثر من معامل نوع، ويمكننا كتابة دالة تقبل نوعين مختلفين ونضعهما في مخزن مؤقت، يمكن استخدام العنصر الأول للحصول على المفتاح والثاني للقيمة مثل:

```
fun <K, V>put(key: K, value: V): Unit
```

ستجد أن هذه الدالة مألوفة لدي أي شخص استخدم الأنواع المُعمَّمَة من قبل في التجميعات (Collections).

## 2. أصناف ذات معاملات غير محدَّدة النوع

ليست الدوال فقط يمكن أن تملك معاملات غير محدَّدة النوع، فالأصناف (أو الأنواع بتعبير أدق) نفسها يمكن أن تملك معاملات غير محدَّدة النوع أيضًا، يشار إلى هذه الأنواع أحيانًا على أنها أنواع حاوية (container types) بسبب ارتباطها الوثيق مع التجميعات (collections) وحقيقة أنها تحتوي على معامل نوع أو أكثر.

نستخدم مرّة أخرى صياغة أقواس الزاوية للتصريح عن نوع ذي معاملٍ غير محدَّد النوع، وهذه المرَّة على الجانب الأيمن من اسم النوع، فعلى سبيل المثال، للتصريح عن النوع Sequence وهو سلسلة عناصر تكون من النوع T، يمكننا كتابة ما يلي:

```
class Sequence<T>
```

مرّة أخرى، يمكننا التصريح عن أكثر من معامل نوع واحد:

```
class Dictionary<K, V>
```

| تنبيه | الأنواع ذات المعاملات غير محدَّدة النوع الأكثر استخدامًا هي التجميعات (collections)، وسنتحدث عن هذا بأكثر تفاصيل في الفصل العاشر، التجميعات. |
|---|---|

عند التصريح عن معامل نوع ضمن صنف، يجب علينا تحديد هذا النوع بالضبط عند إنشائه، لذلك، لإنشاء

نسخة من الصنف Sequence لقيمة منطقيَّة Boolean، نكتب ما يلي:

```
val seq = Sequence<Boolean>()
```

بالنسبة إلى النوع Dictionary، يمكننا فعل شيء مثل هذا:

```
val dict = Dictionary<String, String>()
```

لاحظ أنَّه لا يوجد سبب يشير إلى أنَّ معاملات النوع المختلفة ليس بإمكانها الإشارة إلى نوع حقيقي نفسه، فليس عليها ذلك، وهذا هو الهدف من جعلها مختلفة بالأصل، والخيار متروك للمستخدم أولًا وآخرًا.

## 3. التعددية الشكلية المقيَّدة

تعتبر الدالة مُعمَّمَة النوع مفيدة للغاية، ولكنها محدودة إلى حد ما، ففي كثير من الأحيان سوف نجد أنفسنا راغبةً في كتابة دوال مُعمَّمَة النوع لبعض الأنواع التي تشترك معًا في صفة مميَّزة، على سبيل المثال، قد نرغب في تحديد دالة لإرجاع أدنى قيمة من قيمتين، وذلك للقيم التي تدعم مفهوم الموازنة أو بعضًا منه.

سنبدأ بكتابة دالة تحتوي على معامل نوع يمثِّل النوعين اللذين يجري موازنتهما. ولكن كيف يمكننا موازنة هذه القيم، لأنَّه يمكن أن تكون من أي نوع، بما في ذلك Any نفسه؟ وبما أن النوع Any لا يملك دوالًا تجري عملية الموازنة، فلن تكون هنالك طريقة لموازنة قيمتين من نوعه.

الحل هو تقييد الأنواع لتلك التي تدعم عمليات الموازنة وتوفِّر دوالًا لها، وبهذه الطريقة، سيعرف المصرِّف أنه بغض النظر عن نوع الوسائط وقت التشغيل، يجب أن تكون هذه الدوال متاحة للنوع المستعمل، وبذلك يسمح لنا باستدعائها، ويدعى هذا المفهوم بالتعددية الشكلية المقيَّدة (bounded polymorphism).

### أ. قيود الحدود العليا

تدعم كوتلن أحد هذه الأنواع من الحدود المعروف باسم «قيد الحد الأعلى» (upper bound)، وكما يوحي الاسم، يفرض هذا القيد حدودًا عليا للأنواع لتلك التي هي أصناف فرعيَّة من الحد فقط، ولاستخدام قيد حد أعلى، نصرِّح عنه ببساطة بجانب معامل النوع:

```
fun <T : Comparable<T>>min(first: T, second: T): T {
```

```
        val k = first.compareTo(second)
        return if (k <= 0) first else second
    }
```

النوع Comparable هو أحد أنواع المكتبة القياسيَّة التي تعرِّف الدالة compareTo، والتي توازن بين عنصرين وتُرجع عددًا أصغر من الصفر إذا كان العنصر الأول أصغر من الثاني، وأكبر من الصفر إذا كان العنصر الأول أكبر من الثاني، والعدد صفر إذا كانا متساويان. ففي المثال السابق، عرَّفنا معامل نوع وقيدناه بالحد الأعلى إلى النوع <Comparable<T، لذلك في أي وقت تُستدعَى الدالة min، يجب أن تكون قيمة T مشتقة أو متوسعة من هذا النوع (أي نوعًا فرعيًا منه):

```
val a: Int = min(4, 5)
val b: String = min("e", "c")
```

كما ترى، استطعنا استدعاء الدالة min مع الأعداد الصحيحة والسلاسل النصيَّة، مع إعادة النوع الصحيح مرَّة أخرى، بما أن كلا النوعين، الأعداد الصحيحية Int والسلاسل النصية String يوسّعان (extend) النوع Comparable، ومع ذلك، إذا جربنا استعمال الدالة min مع نوعٍ مثل Pair، والذي لا يوسّع النوع Comparable أي ليس نوعًا فرعيًا منه، فسيبعث المصرِّف خطأً.

| تنبيه | كلما استُخدِم معامل نوع دون قيدٍ لحد أعلى واضح، فسيستخدم المصرِّف النوع Any كحد أعلى ضمني تلقائيًا. |
|---|---|

ومن الأمور المهمة أيضًا أننا لا نستطيع استدعاء الدالة min بتمرير نوعين مختلفين إليها كما فعلنا من قبل مع الدالة random، إذ تذكر أنَّنا إذا استدعينا الدالة random مع معاملات من نوع String، و Int و Boolean، فسيعدُّها المُصرِّف على أنَّها موسّعة من النوع Any ويتحقَّق بذلك قيد الحد الأعلى الافتراضي.

ومع ذلك، إذا كان علينا استدعاء الدالة min مع تمرير سلسلة نصية وعدد صحيح سويةً إليها، على سبيل المثال، فسيكون قي قيد الحد الأعلى للنوع String هو <Comparable<String وقيد الحد الأعلى للنوع Int هو <Comparable<Int، وبما أن كلا النوعين ليس نوعًا عُلْويًا (supertype) للآخر (أنظر مناقشة التباين في

القسم التالي)، فسيكون النوع العلوي المشترك الوحيد الذي يستطيع المصرِّف الاختيار منه هـو النـوع Any، وبما أن هذا الأخير لا ينفِّذ Comparable (أي لا يوسّعه وليس متفرعًا منه)، فلا يمكن استخدامه كنوع لمعامل النوع لأنه لا يستوفي القيود المفروضة، ولذلك لن يملك المصرِّف الخيار سوى إطلاق خطأ.

## قيود متعدِّدة

قد نرغب أحيانًا، في فرض عدّة قيود للحدود العليا، على سبيل المثال، قد نرغب في تطوير مثال دالـة min() لتعمل فقط على القيم القابلة للسلسلة (serializable values) أيضًا، ولفعل هذا، سننقل تصريح قيد الحد الأعلى خارج معامل النوع ونضعه في عبارة where منفصلة:

```
fun <T>minSerializable(first: T, second: T): T
where T : Comparable<T>,T : Serializable {
     val k = first.compareTo(second)
     return if (k <= 0) first else second
}
```

لاحظ أن جميع قيود الحدود العليا مدرجة في تعبير منفصل مع where وتشكِّل بذلك اتحاد قيد الحد العلوي.

الآن، إذا كـان لـدينا نوعًا يحقِّق أحـد القيـدين السـابقين فقـط دون الآخـر، فيطلـق المصـرِّف خطـأ إذا حاولنا استخدامه مع الدالة minSerializable:

```
class Year(valvalue: Year) : Comparable<Year> {
     override fun compareTo(other: Year): Int =
     this.value.compareTo(other.value)
}
```

السطر التالي سيفشل في التصريف:

```
val a = minSerializable(Year(1969), Year(2001))
```

لكن إذا وسـعّنا النـوع Year لينفِّذ Serializable أيضًـا، فيمكننـا اسـتخدامه آنـذاك مـع الدالـة minSerializable:

```
class SerializableYear(valvalue: Int) : Comparable<SerializableYear>,
Serializable {
     override fun compareTo(other: SerializableYear): Int =
     this.value.compareTo(other.value)
}
val b = minSerializable(SerializableYear(1969), SerializableYear(1802))
```

يمكن للأصناف تحديد عدة قيود على الحدود عليا وذلك بالشكل التالي:

```
class MultipleBoundedClass<T>where T : Comparable<T>, T : Serializable
```

لاحظ أن الصياغة مشابه، إذ عبارة `where` مكتوبة بعد معامل النوع.

## 4. تباين النوع

يشير تباين النوع (Type variance) إلى تقنيات وأساليب يمكننا من خلالها السماح أو عدم السماح «بالتعددية الشكلية للأنواع الفرعية» (subtyping) في الأنواع التي تملك معاملات مُعمَّمة النوع (parameterized types) الخاصة بنا، فإذا افترضنا مثلًا أنَّ الصنف `Apple` هو نوع فرعي من الصنف `Fruit`، فهل هذا يعني أنَّ `Crate<Apple>` هو نوع فرعي من `Crate<Fruit>`؟، تظن للوهلة الأولى أنَّ الجواب هو: «أجل، بالتأكيد» بما أنَّ النوع `Apple` يمكن استخدامه في أي مكان يتوقع فيه استخدام النوع Fruit، لكن بشكل عام، الجواب هو لا.

في الواقع، يمكن أن يكون `Crate<Apple>` نوعًا فرعيًا من `Crate<Fruit>`، أو لا اعتمادًا على نوع التباين المستخدم.

### أ. انعدام التباين (اللاتباين)

أولًا، دعنا نناقش لماذا لا يكون `Crate<Apple>` نوع فرعيًا من `Crate<Fruit>` بشكل افتراضي؛ لنبدأ بإنشاء بعض الأصناف:

```
class Fruit
class Apple : Fruit()
```

```
class Orange : Fruit()
class Crate<T>(val elements: MutableList<T>) {
     fun add(t: T) = elements.add(t)
     fun last(): T = elements.last()
}
```

كما ترى، الصنف Crate هو مجرَّد مُغلِّف للنوع MutableList؛ إذا عُرِّفَت دالةٌ تقبل معاملًا باسم creat من النوع Crate<Fruit>، فقد تقرِّر الدالة إضافة عنصر جديد إليه بالشكل التالي:

```
fun foo(crate: Crate<Fruit>): Unit {
     crate.add(Apple()) // لا يُصرَّف
}
```

يبدو هذا جيّدًا لحد الآن، لنفترض أنه لدينا سلة (crate) وتحتوي على فاكهة أي النوع Fruit، ولكن التفاح أي النوع Apple هو من أنواع الفاكهة، لذلك لماذا لا يمكننا إضافته إلى السلة؟ السبب يكمن في التعليمات البرمجيَّة التي تنفذها هذه الدالة، وماذا يحدث بعدها، دعنا نتخيَّل أننا قادرين على كتابة الشيفرة البرمجيَّة التالية:

```
val oranges = Crate(mutableListOf(Orange(), Orange()))
foo(oranges)
val orange: Orange = oranges.last()
```

يبدو هذا جيّدًا، حيث أنَّ التابع foo يطلب سلة من الفاكهة والبرتقال أي المتغيِّر oranges هو من الفاكهة أيضًا، وبالتالي، ينبغي عدم وجود مشكلة إلى الآن؛ على أي حال، الخطأ في هذا التفكير واضح إذ يعلم foo أن ما يمرَّر إليه هو سلة من الفاكهة أي معاملًا من النوع Fruit، ويعتقد أنه يمكن أن يضيف تفاحة إلى تلك السلة والتي هي في الواقع سلة من البرتقال أي من النوع Orange، وبالتالي، إذا كانت هذه الشيفرة البرمجيَّة مسموح بها، فسنحصل على الاستثناء ClassCastException وقت التشغيل عند الوصول إلى السطر الثالث الذي يجلب آخر عنصر أضيف إلى السلة والذي هو تفاحة وليس برتقالة أي من النوع Apple وليس Orange.

إن أبسط حل لهذه المشكلة هو النهج المتَّبع في كوتلن، وهو جعل معامل النوع غير متباين (نهج اللاتباين، أي Invariance) بشكل افتراضي، وعندما يكون كذلك، لا توجد علاقة على مستوى النوع الفرعي (subtype relationship) بين الأنواع، وهذا يعني أن النوع M<T> ليس نوعًا فرعيًا (مولودًا) ولا نوع أعلى (والدًا) للنوع

M<U>، بغض النظر عن العلاقة بين T و U. ولذلك بالنسبة إلى المُصرِّف، فإن Crate<Apple> و Crate<Fruit> مرتبطين مثل Crate<Apple> و Crate<BigDecimal>.

### ب. التغاير

عرفنا ما يعنيه اللاتباين (invariant)، لكن قد يأتي هذا مع بعض المشاكل؛ دعنا نعود إلى مثال السلة crate ونتخيّل وجود دالة أخرى سندعوها verify مهمتها التحقق من أنه إذا كان لدينا سلة فاكهة، فإن كل الفاكهة التي فيها صالحة للأكل، وإذا لم تكن كذلك، فعلينا التخلُّص من كل الفاكهة أو سنخاطر بالإصابة ببعض الأمراض الخطيرة، دعنا نضيف تلك الدالة إلى الصنف Fruit:

```
open class Fruit {
     fun isSafeToEat(): Boolean = ...
}
```

والآن، إذا أردنا كتابة تنفيذ للدالة verify، فسوف نعرِّف الدالة للتعامل مع سلة من الفاكهة، بعد كل شيء، طالما كانت الفاكهة مُعلبَّة ومحفوظة، فهي بالتأكيد صالحة للأكل ولن نقلق حيالها:

```
fun isSafe(crate: Crate<Fruit>): Boolean = crate.elements.all{
     it.isSafeToEat()
}
```

الآن، نحن نعلم بالفعل أن الشيفرة البرمجيّة التالية لن تصرَّف:

```
val oranges = Crate(mutableListOf(Orange(), Orange()))
isSafe(oranges)
```

لكن هذه المرَّة، لا يوجد سبب لعدم تصريفها، فنحن نريد ببساطة استدعاء الدالة isSafeToEat() على كل عنصر من الفاكهة الموجودة في oranges، ولمّا كانت تلك الدالة معرَّفة في صنف Fruit نفسه، فإن كل عنصر من تلك العناصر تمتلكها، أي لديها الدالة isSafeToEat().

تكمن الإجابة في تغيير ما يُعرَف بتباين (variance) الصنف Crate (السلة)، فنحن نرغب في السماح باستعمال سلة من البرتقال (أي النوع Orange) في الموضع الذي يمكن فيه استعمال سلة من الفاكهة (أي النوع

Fruit)، ولكن بأمان؛ وهذا يعني أنَّنا نريد أن يُعدُّ Crate<Orange> نوعًا فرعيًا من Crate<Fruit>، ونحن نعرف بالفعل أن هذا غير آمن عندما نعدّل على سلة (نسخة من Crate) بإضافة عناصر من أنواع فرعية -من النوع Fruit- إليها، مثل التفاح (النوع Apple) أو الإجاص (النوع Pear)، لذا هل هنالك طريقة لفعل ذلك؟ والإجابة هي نعم، وتسمى التغاير (covariance).

عندما تعرّف صنفًا، يمكننا تحديد معامل النوع على أنَّه متغاير، وهذا يعني أن الصنف سيحافظ على العلاقة التي على مستوى الأنواع الفرعية المولودة (subtyping relationship) لمعامل النوع المحدَّد، ولفعل ذلك، نرفق البادئة out بمعامل النوع بالشكل التالي:

```
class CovariantCrate<out T>(val elements: List<T>)
```

تذكر أن قابلية التعديل (mutation) على النسخة هو الذي تسبب لنا من قبل بالمشاكل، لذلك، من أجل السماح باستعمال نهج التغاير، يصرُّ المصرِّف على عدم السماح بتعديل النسخة، فكيف يمكن فرض هذا؟ يمكن فعل ذلك عن طريق التحقق من أن معامل التغاير لا يُستخدَم كدخل للدالة، ولكن إذا لم يُستخدَم كدخل للدالة، فلا يمكن أن تُستخدم التفاح (النوع Apple) موضع استعمال البرتقال (النوع Orange).

الحالة المعاكسة صحيحة، ومع ذلك، لا يزال بإمكاننا استخدام T كقيمة الإرجاع، وهذا لأن أية شيفرة برمجيّة تتوقع أن سلة الفاكهة تحوي فاكهة والتفاح هو أحد أنواع الفاكهة، صحيح! :-). لذلك، لا يكون المعامل T معامل دخلٍ للصنف CovariantCrate، ويجب حذف الدالة add الموجودة في الصنف Create. يمكن أن تبقى الدالة last، إذ تعيد النوع T فقط:

```
class CovariantCrate<out T>(val elements: List<T>) {
     fun last(): T = elements.last()
}
```

يمكننا الآن التحقق من أن الفاكهة صالحة للأكل دون أن ينبس المصرِّف ببنت شفة:

```
val oranges = CovariantCrate(listOf(Orange(), Orange()))
isSafe(oranges)
```

التغاير قوي بشكل لا يصدَّق ويُستخدَم من قبل العديد من أنواع التجميعات غير القابلة للتغيير مثـل: Set، و List و Map، و Iterator و Collection إذ جميعهـا تعـرِّف معامـل النـوع كمتغـاير، بالإضـافة إلى Pair و Triple و Lazy وغيرها الكثير.

### ت. نوع الإرجاع متغاير

يمكن أن يكون نوع الإرجاع للدالة متغايرًا (covariant) أيضًا، وهـذا هـو الافتراضـي، وبالتـالي، إذا كـان نـوعٌ فـرعيٌ يـرغب في إرجـاع نـوع أكـثر تحديـدًا، يمكنـه القيـام بـذلك عن طريـق إعـادة توفـير تعريـف (override definition):

```
open class Animal
class Sheep : Animal()
class Frog : Animal()
abstract class Farm {
     abstract fun get(): Animal
}
abstract class SheepFarm() : Farm() {
     abstract override fun get(): Sheep
}
```

يمكنك أن ترى أن لـدينا هـرم أنـواع الحيوانـات السـابق، وأن النـوع Farm المُعمَّمَ تُرجـع النـوع Animal فقـط، وبشكل أكثر خصوصيَّة، ترغب SheepFarm() بإرجـاع Sheep، وهـذا ممكن في كـوتلن، عنـد اسـتدعاء دالـة، فـإن النوع الذي سيستنتجه المصرِّف لقيمة الإرجاع سيعتمد على نوع المتغير farm نفسه:

```
val farm: Farm = SheepFarm()
val animal1 = farm.get()

val sheepFarm = SheepFarm()
val animal2 = sheepFarm.get()
```

المتغير animal1 هو من النوع Animal، في حين أن المتغيِّر animal2 هو من النوع Sheep.

### ث. التغاير المضاد

التغاير المضاد (contravariant) هو عكس التغاير (covariant)، عندما يوسم معامل نوع بأنَّه متغاير مضاد، فإن العلاقة بين معاملات النوع تنعكس بين معاملات النوع في الأنواع نفسها، أي أن `String` هي نوع فرعي من `Any` لكن `Box<String>` سيكون نوع والد (أعلى) من `Box<Any>` إذا وسم معامل نوع الصنف `Box` على أنَّه متغاير مضاد.

إن أردنا وسم معامل نوع على أنَّه متغاير مضاد، نستعمل الكلمة المفتاحيَّة `in` (عكس `out` إذا لاحظت ذلك).

قد يبدو هذا غريبًا بعض الشيء في البداية، فلماذا نريد عكس العلاقة؟ مثل العادة، سنوضِّح ذلك عن طريق الأمثلة.

تخيَّل أن لدينا صنفًا يدعى `EventStream` يُنتِج أحداثًا من نوع `T`، ويقبل الصنف `EventStream` الخاص بنا مستمعًا (listener) يُستدعَى في كل مرة يُولَّد فيها حدث:

```
interface Listener<T> {
     fun onNext(t: T): Unit
}
class EventStream<T>(val listener: Listener<T>) {
     fun start(): Unit = ...
     fun stop(): Unit = ...
}
```

الآن، إذا أنشأنا `EventStream` من السلاسل النصيَّة، فيمكننا أن نمرِّر إليه مستمعًا من نوع سلسلة نصية أيضًا بالشكل التالي:

```
val stringListener = object : Listener<String> {
     override fun onNext(t: String) = println(t)
}
val stringStream = EventStream<String>(stringListener)
stringStream.start()
```

كل شيء يبدو جيّدًا حتى الآن، دعنا ننشئ مجرًى (stream) آخر، ولكن هذه المرة من

النوع Date:

```
val dateListener = object : Listener<Date> {
     override fun onNext(t: Date) = println(t)
}
val dateStream = EventStream<Date>(dateListener)
dateStream.start()
```

هل شعرت بوجود خطأ ما؟ لقد كتبنا نفس المستمع مرَّتان، وما فعلناه هو تغيير معاملات النوع فقط، ألا يمكننا بدلًا من ذلك كتابة مستمع مسجِّل (logginglistener) واحد واستخدامه بعد ذلك لأي مجرى (stream)؟ فالدالة الوحيدة التي نحتاج إلى الوصول إليها هي toString، وهي معرَّفة في Any نعلم أنَّ String و Date هما بالتأكيد أنواعًا فرعيَّةً من Any:

```
val loggingListener = object : Listener<Any> {
     override fun onNext(t: Any) = println(t)
}
```

إذا حاولنا تطبيق هذا النهج، فإن المصرِّف يرفض استخدام هذا المستمع كوسيط لـ EventStream، وهذا لأن المصرِّف يتوقع <Listener<String، ثم يجب أن يتلقى إما مستمع من هذا النوع، أو مستمع من نوع فرعي، وهذا مجرَّد أحد مبادئ البرمجة الكائنية: يمكن استدعاء دالة بنوع أو نوع فرعي من النوع المطلوب.

وهذه هي الواقعة التي ينقذك فيها التغاير المضاد، وذلك عن طريق وسم معامل النوع بأنَّه متغاير، ومن ثم، لنوع مثل M، يكون <M<T نوعًا فرعيًا من <M<U إذا كان U هو نوع فرعي من T، وفي حالتنا، هذا يعني أن <Listener<Any يعتبر الآن نوعًا فرعيًا من <Listener<String لأن String هو نوع فرعي من Any، ونحن قادرون على استخدامه كما نشاء.

لنحدِّث الصنفين Listener و EventStream لاستعمال نهج التغاير المضاد:

```
interface Listener<in T> {
     fun onNext(t: T): Unit
}
class EventStream<in T>(val listener: Listener<T>) {
```

```
    fun start(): Unit = TODO()
    fun stop(): Unit = TODO()
}
```

يمكننا الآن استخدام هذا النهج لأي نوع نريده:

```
EventStream<Double>(loggingListener).start()
EventStream<BigDecimal>(loggingListener).start()
```

تذكر أنَّه عندما وسمنا معامل النوع على أنَّه متغاير مضاد، قيَّد المصرّف استخدام معامل النوع لإرجاع القيم فقط، وفي المقابل، عند استخدام التغاير المضاد، يمكننا فقط استخدام معاملات النوع كمعاملات إدخال وليس كأنواع إرجاع، والسبب هو في الأساس هو عكس مشكلة `Fruit` و `Orange` التي بدأنا الفصل بها.

إذا كان لدينا دالة تُرجع `T` ولقد سمحنا بأن تكون `T` متغايرًا مضادًا، فقد تتوقع نسخة معيَّنة تأخذ قيمًا من النوع `Orange`، ولكن يمكن أن تُعطَى قيمة من النوع `Fruit`، وستفشل محاولة تقشير البرتقال إذا كانت الفاكهة المعطاة عبارة عن `Tomato`!

يمكننا توضيح ذلك من خلال المثال السريع التالي:

```
interface Generator<in T> {
    fun generate(): T
}

class OrangePicker(val generator: Generator<Orange>) {
    fun pick() {
        val orange = generator.generate()
        peel(orange)
    }
        fun peel(orange: Orange): Unit = // تقشير البرتقال
}
```

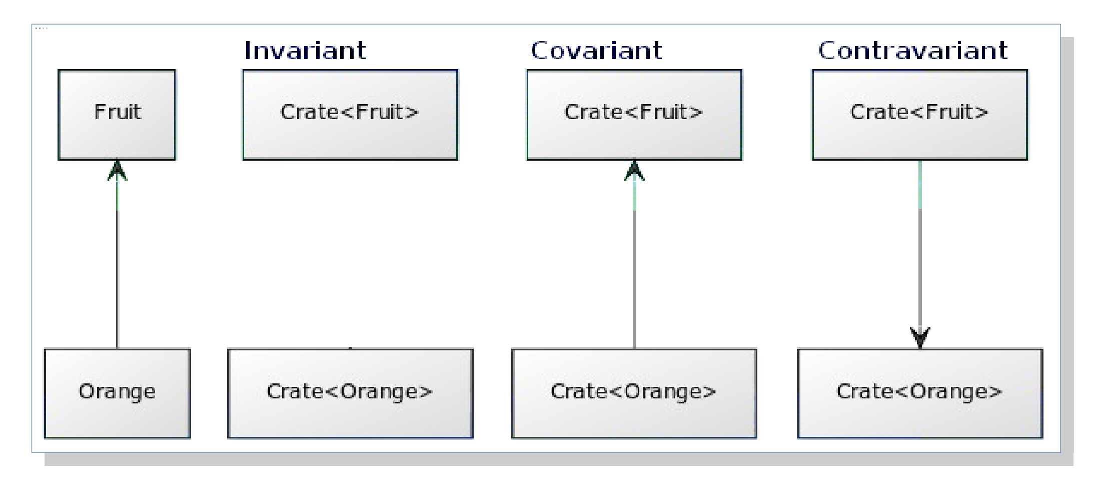
وسمنا الصنف `generator` على أنَّه متغاير مضاد ويُرجع معامل نوع، وعندما ننشئ نسخةً من `OrangePicker`، سنمرِّر `generator` إليها؛ لنفترض أننا استدعيناه مثل هذا، مع مولِّد `Fruit` مُعمَّم:

```
val generator = object : Generator<Fruit> {
     override fun generate(): Fruit = Tomato() // random fruit
}
val picker = OrangePicker(generator)
picker.pick()
```

بما أنَّ generator متغاير مضاد، يعدُّ مولِّد الفاكهة (أي generator المستعمل لتوليد الفاكهة) نوعًا فرعيًا من مولِّد البرتقال (أي generator المستعمل لتوليد البرتقال) وهكذا يمكن تمريره إلى الباني، ومع ذلك، فإن دالة pick تتوقَّع نسخ من Orange، ولذلك ستنتهي برمي بالحصول على استثناء يُرمَى وقت التشغيل، ولهذا السبب، يقتصر التغاير المضاد على مواقف الإدخال فقط.

## ج. نظرة عامة على التباين

يوضح المخطط التالي التباينات المختلفة والعلاقات بين الأصناف، وتذكر أنَّه إذا كان النوع Orange هو نوع فرعي من النوع Fruit، فإن الحالة الافتراضيَّة هي لاتباين (invariant) أي لا يوجد بين سلة البرتقال (النوع Crate<Orange>) وسلة الفاكهة (Crate<Fruit>)ـ أية علاقة. وباستخدام التغاير، فإن Crate<Orange> هو نوع فرعي من Crate<Fruit>، وأخيرًا، مع التغاير المضاد، فإنَّ النوع Crate<Orange> هو نوع والد (أعلى) من النوع Crate<Fruit>.



## 5. النوع Nothing

تطرقنا في الفصل الثاني، أساسيات كوتلن، باختصار إلى التسلسل الهرمي للأنواع في كوتلن، ولقد ذكرنا النـوع Nothing: هو نوع فرعي وليد لجميع الأنواع الأخرى مثل أن Any هـو الصـنف الأعلى الوالـد لجميـع الأنـواع. فكـرة النوع Nothing ليست جديدة لأشخاص الذين استخدموا لغة برمجة وظيفيّة (functional language) مثـل سكالا، وبالنسبة للآخرين التي تبدو لهم الفكرة جديدة، سنخبرهم لماذا هذا النوع مفيد.

حالة الاستخدام الأولى للنوع Nothing هي للإشارة إلى أنَّ الدالة لن تكتمل بشـكل طـبيعي، ومـا نعنيـه بكلمـة «طبيعي» أنَّه لا يتوقَّع منها إرجاع قيمة، أي أنَّها قـد تقـوم بحلقـة لا نهائيَّة عن قصـد، وتنتهي فقـط عنـدما يتم إنهـاء العملية أو الخيط، أو قـد تتوقـف عنـد رمي اسـتثناء، فعلى سـبيل المثـال، الدالـة error المعرَّفـة في مكتبـة كـوتلن القياسيَّة لديها التنفيذ التالي:

```
inline fun error(message: Any): Nothing = throw
IllegalStateException(message.toString())
```

ومع ذلك، يكون الاستخدام الرئيسي هو معامل نوع في الأنواع المتباينة (variant types)، وإذا كـان لـدينا نوعًـا متغـايرًا (covariant type) ونـرغب في إنشـاء نسـخة متوافقـة مـع جميـع الأنـواع الأعلى الوالـدة ( supertypes)، فيمكننا استخدام المعامل Nothing كمعامل نوع، فعلى سبيل المثال، انظر إلى النوع التالي:

```
class Box<out T>
```

سيكون المثال التالي متوافقًا مع هذا:

```
Box<Nothing>()
```

قد يبدو من غـير المجـدي أن يكـون لـديك Box من Nothing (أي صـندوق فـارغ من لا شـيء)، لكنـه سـيعمل جيّدا كصندوق فارغ (empty box)، فالصندوق الفارغ لا يحتـوي على عناصـر، لـذا فـإن الـدوال الـتي تُرجـع نـوع Nothing ليس فيها خلل، وإذا كان الصندوق الفارغ غير قابـل للتغيـير أيضًـا، فسـنحتاج إلى نسـخة واحـد، وهكـذا، على سبيل المثال، نعرِّف قائمة فارغة في كوتلن ككائن منفرد مشترك ينفِّذ النوع List<Nothing>.

يُستخدَم النوع Nothing كنوع في العادة لهذه الحيلة عندما يكون لـدينا نـوع قـد نـرغب بـأن يكـون فارغًـا أو

يجري عملية فارغة no-op، لنفترض أن لدينا النوع `Marshaller` والذي عمَّمٌ معامله ليشمل نوع الرسالة التي يعيدها، وهو شيء من هذا القبيل:

```
interface Marshaller<out T> {
     fun marshall(json: String): T?
}
```

يمكننا بسهول إنشاء نسخة لعملية فارغة (no-op) يمكن استخدامها في أي مكان يتوقَّع فيه استخدام `Marshaller`:

```
object NoopMarshaller : Marshaller<Nothing> {
override fun marshall(json: String) = null
}
```

وأخيرًا، تجدر الإشارة إلى أنه ليس هنالك نسخ من `Nothing`، فهو معرَّف على أنَّه نوع، لكن لا يمكن استنساخ نسخة منه.

## 6. الأنواع المُسقَطة

في قسم تباين النوع، عملنا من خلال أمثلة على التغاير والتغاير المضاد، وكيف تُستخدم معاملات النوع المقيَّدة كأنواع المدخلات أو أنواع إرجاع على التوالي، وهذه ليس مشكلة في العادة عندما نحدِّد واجهاتنا وأصنافنا إذ نستطيع الحصول على التجريدات الصحيحة المطلوبة.

ولكن ماذا عن الحالة التي قام فيها شخص آخر بتعريف الصنف ليكون لامتباين (invariant) وأنت تطلب استخدامه بنهج التغاير أو التغاير المضاد؟ تعالج كوتلن هذا عن طريقة تقديم إضافة قويّة تسمى «الأنواع المُسقَطَة» (Type Projections).

عند استخدام معاملات النوع، يوجد تمييز بين تباين موقع الاستخدام وتباين موقع التصريح (الإعلان)، فتباين موقع الاستخدام (Use site variance) هو المصطلح المستخدم عندما يحدَّد تباين معاملات النوع بواسطة المتغيِّر نفسه، كما هو الحال في جافا، أمَّا تباين موقع الإعلان هو المصطلح المستخدم عندما يحدِّد النوع أو الدالة التباين، كما في كوتلن.

تسمح لنا الأنواع المُسقطَة بتحديد التباين في موقع الاستخدام بدلًا من ذلك. دعنا نعود إلى مثال سلة الفاكهة القابلة للتغيير:

```
class Crate<T>(val elements: MutableList<T>) {
     fun add(t: T) = elements.add(t)
     fun last(): T = elements.last()
}
```

إذا كنت تتذكّر، كان السبب في تقييد Crate هو أنّنا لم نتمكن من تمرير سلة من البرتقال أي Crate<Orange> إلى دالة تتطلّب سلة من الفاكهة أي Crate<Fruit>، ولقد عملنا حول هذا عن طريق إنشاء صنف جديد، يدعى CovariantCrate، والذي يوسم T على أنّه متغاير، وحذفنا الدالة add لأنّها تستخدم النوع T كنوع للمدخلات.

كان هذا جيّدًا لأنّنا كنا مؤلفي الصنف Crate ويمكننا تكييفه حسب رغبتنا، وإذا كان الصنف يأتي من مكتبة ما، فلن نتمكن من إعادة تعريفه، لذا يمكننا إنشاء صنف جديد ونسخ العناصر أو بدلًا من ذلك استخدام مفهوم الأنواع المُسقطَة.

يسمح لنا مفهوم النوع المُسقَط بتقييد الدوال المتاحة على النوع حتى تفي بالمعايير اللازمة لتعدّ على أنّها متغايرة أو متغايرة مضادة، وإذا استطعنا إبلاغ المصرِّف بأنه ليس لدينا رغبة باستدعاء add على النوع Crate، فلا يوجد سبب يمنعنا من استخدامها وفق نهج التغاير.

لفعل ذلك، فإننا نستخدم الكلمتين المفتاحيتين out و in عندما نعرِّف الدالة التي ستقبل معاملات مُعمَّمة النوع؛ فعلى سبيل المثال، لإنشاء دالة تقبل النوع Crate المسقط على أنّه متغاير، يمكننا تعريف ما يلي:

```
fun isSafe(crate: Crate<out Fruit>): Boolean = crate.elements.all{
     it.isSafeToEat()
}
```

يمكننا الآن استدعاء الدالة isSafe مع النوع Crate الأصلي اللامتباين:

```
val oranges = Crate(mutableListOf(Orange(), Orange()))
isSafe(oranges)
```

إذا كنت تتذكّر، فقد فشل السطر الأخير سابقًا، قبل أن نعرف الأنواع المُسقطَة للأسف.

تعمل نفس الحيلة للأنواع المُسقطة ذات التغاير المضاد؛ فإذا استخدمنا مثال تجرى الحدث السابق، فكان من الممكن أن نستخدم مستمعًا لامتباين عن طريق ملائمة الصنف `EventSteam` لإسقاط النوع `Listener`:

```
interface Listener<T> {
     fun onNext(t: T): Unit
}
class EventStream<in T>(val listener: Listener<in T>) {
     fun start(): Unit = TODO()
     fun stop(): Unit = TODO()
}
```

لاحظ الكلمة المفتاحيَّة `in` المضافة إلى معامل `listener` في الباني.

عند استخدام النوع المُسقَط، يقيدنا المصرِّف باستدعاء الدوال التي تملك معاملات النوع المسموح بها فقط، لذا، إذا أسقطناه على أنَّه متغاير، فيمكننا فقط استدعاء الدوال التي تُرجع `T` (أو التي لا تستخدم `T` على الإطلاق)، وإذا أسقطناه على أنه متغاير مضاد، فيمكننا فقط استدعاء الدوال التي تقبل `T` (أو التي لا تستخدم `T`).

## 7. إزالة الأنواع

صمّمت كوتلن بشكل أساسي لتكون لغةً لآلة جافا الافتراضيّة (JVM)، وعند تصميم JVM للمرَّة الأولى، لم يضمَّن فيها ميزة الأنواع المُعمَّمَة، وبمرور الوقت، شكَّل هذا كان عيبًا كبيرًا في اللغة، لذا، في الإصدار 1.5 من جافا (أو Java SDK 5) الذي صدر عام 2004، أضيفت ميزة الأنواع المُعمَّمَة إلى المصرِّف.

ومع ذلك، بسبب الرغبة في إبقائه متوافقًا مع الإصدارات السابقة من جافا، قرر مصممو جافا أن الأنواع المُعمَّمَة ستطبَّق باستخدام تقنيَّة تسمى «الإزالة» (Erasure) وهو الاسم الذي يطلق على العمليَّة التي من خلالها يزيل المصرِّف معامل النوع أثناء التصريف.

في جافا، سيُصرَّف صنفٌ معرَّف على أنه `List<T>` في الشيفرة المصدرية إلى `List` ببساطة، أو `List<Object>` إذا أردت، وهذا يطرح مشاكل قد عرفنا بعضها بالفعل:

- ستتعارض الدوال التي تملك نفس الأسماء ونفس المعاملات المزالة (erased parameters)،

فستملك كل من `fun print(list: List<String>)` و `fun print(list: List<Int>)` نفس بصمة الدالة بعد الإزالة.

- وقت التشغيل، لا يمكنك رؤية معاملات النوع التي أُستخدمَت عند إنشاء الكائن.
- لا يمكنك اختبار ما إذا كانت النسخة من نوع T.
- لا يمكنك اختبار ما إذا كانت النسخة هي من نوع يملك معاملات معمَّمة.
- لا يمكنك الوصول إلى الصنف `literal` من أجل T.
- اســتبدل بمعامــل النــوع في الأصــناف الــتي تســتخدمه نــوع كــائن أو النــوع المُحــدَّد بقيــد الحد الأعلى.

بما أنَّ كـوتلن تسـتهدف JVM، فـإنَّ كـوتلن مقيَّد أيضًـا بهـذه المشـكلات، يمكننـا رؤيـة هـذا من خلال النظر إلى بايتكود عند إنشاء دالتين متطابقتين تمامًا باستثناء أنَّ كل واحدة منهما تقبل نوعًا مختلفًا من `List`:

```
fun printInts(list: Set<Int>): Unit {
     for (int in list) println(string)
}
fun printStrings(list: Set<String>): Unit {
     for (string in list) println(string)
}
```

هذا هو بايتكود الباني من الدالة الأولى:

```
0: aload_0
1: ldc #9 //String list
3: invokestatic #15
6: aload_0
7: invokeinterface #21, 1 //InterfaceMethod java/util/Set.iterator:
()Ljava/util/Iterator;
12: astore_2
13: aload_2
14: invokeinterface #27, 1 //InterfaceMethod java/util/Iterator.hasNext:
()Z
```

```
19: ifeq
46
22: aload_2
23: invokeinterface #31, 1 //InterfaceMethod java/util/Iterator.next:
()Ljava/lang/Object;
28: checkcast #33 //class java/lang/Number
31: invokevirtual #37 //Method java/lang/Number.intValue:()I
34: istore_1
35: nop
36: getstatic #43 //Field java/lang/System.out:Ljava/io/PrintStream;
39: iload_1
40: invokevirtual #49 //Method java/io/PrintStream.println:(I)V
43: goto 13
46: return
```

وهذا هو بايتكود المتطابق تقريبًا للدالة الثانية:

```
0: aload_0
1: ldc #9 //String list
3: invokestatic #15
6: aload_0
7: invokeinterface #21, 1 //InterfaceMethod java/util/Set.iterator:
()Ljava/util/Iterator;
12: astore_2
13: aload_2
14: invokeinterface #27, 1 //InterfaceMethod java/util/Iterator.hasNext:
()Z
19: ifeq 43
22: aload_2
23: invokeinterface #31, 1 //InterfaceMethod java/util/Iterator.next:
()Ljava/lang/Object;
28: checkcast #55 //class java/lang/String
31: astore_1
32: nop
33: getstatic #43 //Field java/lang/System.out:Ljava/io/PrintStream;
36: aload_1
```

```
37: invokevirtual #58 //Method
java/io/PrintStream.println:(Ljava/lang/Object;)V
40: goto 13
43: return
```

لاحظ أن شيفرة تنفيذ العمليات (opcodes) التي ولدها المصرِّف متشابهة، باستثناء التعليمة 28، والتي هي عملية تحويل للنوع، فالمصرِّف يضيف عملية تحويل لنوع وقت التشغيل لكائن نزولًا إلى النوع وقت التصريف، والفرق الوحيد هو النوع المُحوَّل، والذي هو مخزّن في المصرِّف الساكن (constant pool) ويشار إليه بالرقم بعد عملية التحويل.

تعادل شيفرة البايتكود المولَّدة الشيفرة البرمجيَّة التاليَّة:

```
fun printInts(list: Set<Any>): Unit {
      for (obj in list) {
            println(obj as Int)
      }
}
fun printStrings(list: Set<Any>): Unit {
      for (obj in list) {
            println(obj as String)
      }
}
```

الدوال ليست المشكلة الوحيدة، فإذا كان لدينا صنف يستخدم معامل نوع في بصمة الدالة، فسيُستبدَل بمعامل النوع هذا النوع `java.lang.Object` أو بالنوع المحدَّد في قيد الحد الأعلى إذا كان معامل النوع مقيَّدًا، على سبيل المثال:

```
fun <T : Comparable<T>>max(list: List<T>): T {
      var max = list.first()
      for (t in list) {
            if (t >max)
                  max = t
```

```
    }
    return max
}
```

عند عرض البايتكود، ستلاحظ أنَّه يجب على جسم الدالة تحويل أنواع عناصر القائمة:

```
0: aload_0
1: ldc #9 //String list
3: invokestatic #15
6: aload_0
7: invokestatic #68
10: checkcast #70 //class java/lang/Comparable
13: astore_1
14: aload_0
15: invokeinterface #73, 1 //InterfaceMethod java/util/List.iterator:
()Ljava/util/Iterator;
20: astore_3
21: aload_3
22: invokeinterface #27, 1 //InterfaceMethod java/util/Iterator.hasNext:
()Z
27: ifeq 56
30: aload_3
31: invokeinterface #31, 1 //InterfaceMethod java/util/Iterator.next:
()Ljava/lang/Object;
36: checkcast #70 //class java/lang/Comparable
39: astore_2
40: aload_2
41: aload_1
42: invokeinterface #77, 2
47: iconst_0
48: if_icmple 53
51: aload_2
52: astore_1
53: goto 21
56: aload_1
57: areturn
```

هنالك طريقتان للتعامل مع بعض هذه المشكلات، الأولى، كما تحدثنا في الفصل المتعلِّق بالتوصيفات (annotations)، إذ يمكننا وسم الدوال التي لها نفس البصمة المزالة باسم مختلف عند تصريفها. تذكَّر أننا نفعل ذلك باستخدام التوصيف `@JvmName`.

الطريقة الأخرى شكل محدود من إعادة التأهيل (reification).

## 8. تجسيد النوع

النوع القابل للتجسيد (reifiable type) هو الاسم الذي يُطلَق على النوع عندما يمكن فحص معلوماته وقت التشغيل، وأمثلة على هذا هي الأنواع الغير مُعمَّمَة مثل `String` أو `BigDecimal`، وفي JVM الأنواع الأولية مثل `boolean` أو `double` هي أنواع قابلة للتجسيد.

النوع غير القابل للتجسيد (non-reifiable type) هو الذي عانى من تأثير مزيل النوع (type erasure)، لذلك، بعض أو كامل معلومات نوعه ضاعت وقت التشغيل، ومثال ذلك، الأنواع ذات المعاملات المُعمَّمة (parameterized types) مثل `List<String>` و `List<Boolean>` التي تبدو متشابهة وقت التشغيل.

لقد رأينا كيف أن مزيل النوع يحذف الأنواع وقت التشغيل منعًا لتسببها لمشاكل أشرنا إلى بعضها آنفًا، والآن سننظر إلى الطريقة التي يمكننا بها حل بعض هذه المشاكل، فلقد قدَّمت كوتلن ميَّزة تسمى تجسيد النوع (type reification) التي تحافظ على معلومات النوع وقت التشغيل من أجل الدوال المضمنة (type reification).

لاستخدام هذه الميَّزة، نضيف الكلمة المفتاحية `reified` قبل معامل النوع، كما هو موضّح في المقتطف التالي، ومن ثم نحن قادرون على القيام بعمليات على النوع T:

```
inline fun <reified T>runtimeType(): Unit {
    println("My type parameter is " + T::class.qualifiedName)
}
```

لاحظ كيف يمكننا الحصول على نوع وقت التشغيل من أجل T على شكل صنف `KClass`، ويمكننا أيضًا استخدام T للتحقُّق من النوع:

```
inline fun <reified T>List<Any>.collect(): List<T> {
```

```
      return this.filter { it is T }.map { it as T }
}
```

في هذا المثال، نعمل على ترشيح عناصر قائمة `List`، ولن نُرجع سوى التي تتطابق مع معامل النوع المعطى، ونفعل ذلك عن طريق التحقُّق من كل عنصر لمعرفة ما إذا كان نسخة من النوع `T`، ونستطيع القيام بذلك فقط إذا كان بإمكاننا الوصول إلى معامل النوع وقت التشغيل لأنه من المستحيل معرفة العناصر التي قد تحتوي عليها القائمة `List` مقدمًا. يمكننا استعمال ذلك بالطريقة التالية:

```
val list = listOf("green", false, 100, "blue")
val strings = list.collect<String>()
```

إذن كيف تقوم كوتلن بهذه الخدعة؟ تكمن الإجابة في حقيقة أن الدوال المُجسَّدة (reified functions) يجب أن تعرَّف على أنَّها `inline`، في جميع الأماكن التي تستدعى فيها الدالة، إذ سيُنسَخ الجسم إلى موقع الاستدعاء، وبما أن المصرِّف يعرف المعامل النوع المستخدم في موقع الاستدعاء، فهو قادر على استبدال مراجع إلى النوع الصحيح بالمرجع الذي يشير إلى `T`.

يمكننا أن نرى هذا من خلال فحص شيفرة البايتكود للدالة المُجسَّدة، دعنا نُعرِّف مثالًا أقصر، حتى يكون لدينا بايتكود أقصر يمكننا فحصه:

```
inline fun <reified T>printT(any: Any): Unit {
      if (any is T)
      println("I am a tee: $any")
}
```

تتحقّق هذه الدالة من أن معامل الإدخال من النوع `T`، وإذا كان الأمر كذلك، سنطبعه، وسوف نستدعيه بالشيفرة البرمجيّة التاليّة:

```
printT<Int>(123)
```

سيبدو البايتكود المولَّد كالتالي:

```
0: aload_0
```



```
1: ldc #159 //String args
3: invokestatic #163
6: bipush 123
8: invokestatic #125 //Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
11: astore_1
12: nop
13: aload_1
14: instanceof #122 //class java/lang/Integer
17: ifeq 48
20: new #8 //class java/lang/StringBuilder
23: dup
24: invokespecial #11 //Method java/lang/StringBuilder."<init>":()V
27: ldc #150 //String I am a tee:
29: invokevirtual #17
32: aload_1
33: invokevirtual #153
36: invokevirtual #40 //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
39: astore_2
40: nop
41: getstatic #46 //Field java/lang/System.out:Ljava/io/PrintStream;
44: aload_2
45: invokevirtual #52 //Method
java/io/PrintStream.println:(Ljava/lang/Object;)V
48: return
```

التعليمة 14 هي الأهم هنا، هذا هو السطر الذي يتحقق فيه المصرِّف من إعادة تأهيل معامل نوع، ويشير السطر ذي التعليق ‎#122‎ إلى `java.lang.Integer` في المصرِّف الساكن (constant pool)، والذي كان النوع الصحيح الذي استخدمناه عند استدعاء الدالة.

## 9. قيود النوع العوديَّة

قيود النوع العوديَّة (Recursive type bounds) هو اسم جميل وجذاب ومعقَّد بعض الشيء، لكن سهل شرحه، فهو امتداد للتعددية الشكلية المقيَّدة، ويصف نوعًا يحوي معامل نوع واحدٍ أو أكثر، إذ نوع أحد هذه

المعاملات على الأقل هو النوع نفسه، معقدة أليس كذلك :-D.

كل شيء سيصبح أكثر وضوحًا مع مثال؛ لنفترض أننا نكتب واجهة برمجة تطبيقات API لحسابات في نظام مالي، ونريد تعريف كائن باسم Account، ونريد فرز جميع أنواع الحسابات، ولكن مع تقييد أننا نرغب بفرز الحسابات من نفس النوع، الخطوة الأولى لفعل هذا هي تعريف واجهة لحساباتنا، مع كتابة تنفيذين لها:

```
interface Account {
     val balance: BigDecimal
}
data class SavingsAccount(override val balance: BigDecimal,val
interestRate: BigDecimal) : Account, Comparable<SavingsAccount> {
     override fun compareTo(other: SavingsAccount): Int =
     balance.compareTo(other.balance)
}
data class TradingAccount(override val balance: BigDecimal, val margin:
Boolean) : Account, Comparable<TradingAccount> {
     override fun compareTo(other: TradingAccount): Int =
     balance.compareTo(other.balance)
}
```

لا يوجد شيء جديد في الشيفرة البرمجيّة السابقة، كلا التنفيذين يوسعان الواجهة Account ويوسعان أيضًا النوع Comparable إذ يمكن استخدامه من قبل دوال الفرز في المكتبة القياسيّة، فعلى سبيل المثال، يمكننا الآن موازنة حساب توفير بآخر، ونفس الشيء لحساب التداول:

```
val savings1 = SavingsAccount(BigDecimal(105), BigDecimal(0.04))
val savings2 = SavingsAccount(BigDecimal(396), BigDecimal(0.05))
savings1.compareTo(savings2)

val trading1 = TradingAccount(BigDecimal(211), true)
val trading2 = TradingAccount(BigDecimal(853), false)
trading1.compareTo(trading2)
```

والأفضل من ذلك، أن هذا لم يعمل:

```
savings.compareTo(trading) compile error
```

هذا ما نريد الوصول إليه، ففي هذه الحالة، جزء من هدفنا الأصلي هو السماح بالفرز فقط بين الحسابات التي من نفس النوع.

لاحظ أنَّنا نكرِّر الشيفرة البرمجيَّة في دالتي compareTo، في الواقع هما متطابقين، فهما بسيطان إلى حد ما، لذلك لا توجد الكثير من المشاكل، لكن في النظام الحقيقي قد يمتد هذا على عدة أسطر، هل يمكننا تحسين هذا عن طريق جلب دالة compareTo لتعمل في الواجهة؟

```
interface Account2 : Comparable<Account2> {
     val balance: BigDecimal
     override fun compareTo(other: Account2): Int =
     balance.compareTo(other.balance)
}

class SavingsAccount2(override val balance: BigDecimal) : Account2

class TradingAccount2(override val balance: BigDecimal) : Account2
```

لقد أصبحت المشكلة أنه يمكننا موازنة نسخة من Account2 مع أي نسخة أخرى، فعلى سبيل المثال، ستعمل الشيفرة التالية بدون مشاكل:

```
val savings = SavingsAccount2(BigDecimal(105), BigDecimal(0.04))
val trading = TradingAccount2(BigDecimal(210), true)
savings.compareTo(trading)
```

الآن، قد يتسبب ذلك في مشاكل إذا لم نكن نرغب في موازنة أنواع مختلفة من الحسابات، وهذا هو أحد الآثار الجانبيّة لحقيقة أننا حدَّدنا Account لتنفيذ Comparable<Account>، كيف يمكننا مشاركة تنفيذ الدالة compareTo لكن في نفس الوقت نتجنَّب موازنة أنواع مختلفة؟ قد نفكِّر في معامل النوع، واستخدامه في عبارة extends للنوع Comparable:

```
interface Account3<E> : Comparable<E> {
```

```
        val balance: BigDecimal
        override fun compareTo(other: E): Int =
        balance.compareTo(other.balance)
}

data class SavingsAccount3(override val balance: BigDecimal, val
interestRate: BigDecimal) : Account3<SavingsAccount3>

data class TradingAccount3(override val balance: BigDecimal, val margin:
Boolean) : Account3<TradingAccount3>
```

يبدو هذا مثاليًا في للوهلة الأولى، كل نوع حقيقي (concrete type) سيكون قابلًا للموازنة مع نفسه، وهنالك منطق في الواجهة، ومع ذلك، هنالك مشكلة أن المصرّف لا يتعرّف على `other.balance` في الشيفرة البرمجيّة الخاصة بالموازنة.

بما أن `Account3` يعرّف معامل النوع دون قيود، فيمكننا إنشاء حساب يستخدم `String`، أو `Foo` أو أي شيء نرغب به، وبما أن خاصيّة `balance` ليست معرّفة على `String` أو `Foo`، فسيرمي المصرّف خطأ عندما نحاول الوصول إليه.

إذن كيف يمكننا الحفاظ على تقدمنا وجعل الشيفرة تعمل؟ الجواب هو توفير قيد الحد الأعلى (upper bound) على معامل النوع للنوع `Account` إذ عندما ننشئ نسخة من `Account`، تمرّر واجهة `Comparable` للنوع مع احتوائه على الخاصيّة `balance`، ويمكننا القيام بذلك باستخدام ما يُعرف باسم قيد النوع العودي (recursive type bound):

```
interface Account4<E : Account4<E>> : Comparable<E> {
        val balance: BigDecimal
        override fun compareTo(other: E): Int =
        balance.compareTo(other.balance)
}

data class SavingsAccount4(override val balance: BigDecimal, val
interestRate: BigDecimal) : Account4<SavingsAccount4>
```

```
data class TradingAccount4(override val balance: BigDecimal, val margin: Boolean) : Account4<TradingAccount4>
```

لاحظ كيف يصرّح عن E بتوسيع `Account<E>`، وهذا يعطينا شيفرة `compareTo` البرمجيّة ولا يسمح للحسابات بالموازنة إلا مع أنفسها:

```
val savings1 = SavingsAccount4(BigDecimal(105), BigDecimal(0.04))
val savings2 = SavingsAccount4(BigDecimal(396), BigDecimal(0.05))
savings1.compareTo(savings2)

val trading1 = TradingAccount4(BigDecimal(211), true)
val trading2 = TradingAccount4(BigDecimal(853), false)
trading1.compareTo(trading2)
```

وكما هو الحال في التصميم الأصلي، فإننا سنجد خطأ في التصريف إذا حاولنا فعل التالي:

```
savings.compareTo(trading) compile error
```

**تنبيه**

هنالك عيب نهائي، إذ أننا لا نستطيع إيقاف الحسابات التي عُرِّفت بمعامل نوع حساب آخر، مثل:

```
class BettingAccount: Account<ShareAccount>
```

وهذا القيد موجود في جافا أيضًا.

## 10. أنواع البيانات الجبريّة

أنواع البيانات الجبريّة (Algebraic data types) هي واحدة من مفاهيم البرمجة الوظيفيّة التي تبدو معقّدة في البداية، ولكن سهلة جدًا بالاطلاع على مثال أو اثنين عنها، ويشير المصطلح نفسه إلى حقيقة أن الجبر يعرّف مجموعة من الأشياء والعمليات المسموح بها على هذه الأشياء، على سبيل المثال، في الرياضيات، العامل + معرّف على عددين صحيحين لإرجاع مجموعهما، وبالتالي فهو مفهوم جبري.

لذلك، يعرّف الجبر للنوع العمليات والدوال على هذا النوع، وهنا ظهر مصطلح «نوع البيانات الجبريّة»، في

لغات الحوسبة، يتم تطبيق المصطلح بشكل عام على مجموعة مغلقة من الأنـواع المركبـة (composite types) إذ تقوم تلك الأنواع بتنفيذ الدوال المطلوبة؛ ففي كوتلن، نحقِّق الخاصيَّة المُغلَقة (closed property) باستخدام الكلمة المفتاحيَّة sealed، والتي تقيِّد الأنواع المسموح بها فقط لتلك المعرّفة في نفس الملف.

| تنبيه | يجب عدم الخلط بين اختصار Algebraic Data Types الذي هو ADT (أنواع البيانات الجبرية) وبين اختصار Abstract Data Types الذي هو ADT أيضًا (أنواع البيانات المجرَّدة)، إذ يملكان نفس الاختصار باللغة الأجنبية، إلا أنهما مفهومان منفصلان تمامًا. |
|---|---|

دعنـا ننتقـل من هـذا الكلام النظـري إلى مثـال عملي؛ سنسـتخدم مثـالًا شـائعًا لقائمـة مترابطة (linked list)، الهيكل العام لقائمة مترابطة هو أن كل عنصر في القائمة يرتبط بعقدة (node)، وتحتوي كل عقدة على رابط للعقدة التاليَّة، ويسمح لنا رابط في العقدة الأولى بالتنقل في القائمة بأكملها من خلال المرور عبر الروابط.

يمكننا البـدء بتعريـف صـنف مُغلـق أو مختـوم (sealed) الـذي سيحتوي على عملياتنـا، لاحـظ أنَّه لا يمكن استخدام الكلمة المفتاحية sealed على واجهة مثل:

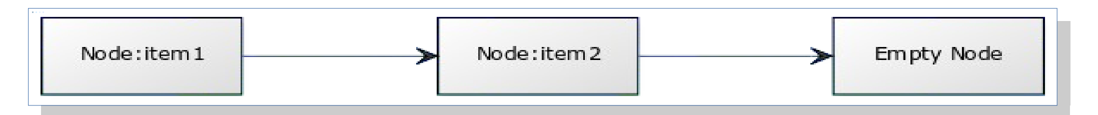
```
sealed class List<out T>
```

بعد ذلك، سنعرّف تطبيقيَن، سيمثِّل الأول العقدة التي تحتوي على القيمة وسيمثّل الثاني عقدة فارغة:

```
class Node<T>(val value: T, val next: List<T>) : List<T>()
object Empty : List<Nothing>()
```

يمكن عدُّ القائمة كسلسلة من العقد تنتهي دائمًا بعقدة فارغة، ولذلك، فإن القائمة الفارغة ليست أكـثر من مجـرد عقدة فارغة.

تملك عقد البيانات خاصيتيّن، واحدة للقيمة التي تحتوي عليها والأخرى للإشارة إلى العنصر التالي في القائمـة. فعلى سبيل المثال، ستبدو قائمة من عنصرين كالتالي:



لاحظ العقدة الفارغة تعرَّف على أنَّها كائن، وهذا لأنها لا تملك حالة، ولذلك نحن لسنا بحاجة إلى أكثر من نسخة واحدة، وبما أننا نستخدم عقدة فارغة واحدة، فيجب استعمال النوع `Nothing` مع معامل النوع للأسباب الموضَّحة في قسم النوع `Nothing`.

لنبدأ بملء الدوال التي تتطلبها القائمة النموذجيَّة؛ يمكننا البدء بالأسهل: هل القائمة فارغة أم لا؟

```
sealed class List<out T> {
     fun isEmpty() = when (this) {
          is Node -> false
          is Empty -> true
     }
}
```

لاحظ أننا نتحقَّق من النوع، وبالاعتماد على نوع القائمة، يمكننا معرفة ما إذا كانت فارغة أو لا، ولا نحتاج حتى للنظر إلى أي من الخاصيات.

وهذا أيضا هو المكان سيظهر استخدام الكلمة المفتاحيَّة `sealed`، فعندما تُستعمَل مع صنف، فسيعرف المُصرِّف أن جميع التنفيذات المكتوبة يجب أن تكون في نفس الملف، وهكذا يعرف مجموعة الأنواع الحقيقية كاملة، ولذلك، عند استخدام صنف مُغلق (sealed class) في تعبير `when`، فإنَّ المُصرِّف قادر على إصدار خطأ إذا لم نغطي جميع الحالات. إذا كان لدينا على سبيل المثال، نوع بيانات جبريّة متكونة من ستّة أنواع ولقد غطينا خمسة في تعبير `when`، سنحتاج إما إلى إضافة النوع الأخير أو عبارة `else`.

بعد ذلك سنضيف دالة `size()`، وسنعرّف هذا بشكل عودي (recursively)، والذي لن يكون فعَّالا على أرض الواقع، لكننا نوضِّح مفهوم أنواع البيانات الجبريَّة وليس فعاليتها:

```
fun size(): Int= when (this) {
     is Node -> 1 + this.next.size()
     is Empty -> 0
}
```

على غرار دالة isEmpty، نتحقق من النوع، فإذا كانت العقدة فارغة، نُرجع صفرًا، أو نضيف واحدة إلى حجم الذيل.

الكثير من الدوال متشابهة في التنفيذات الموفَّرة، فعلى سبيل المثال، ستبدو الدالة head التي تُرجع أول عنصر من القائمة كالتالي:

```
fun head(): T = when (this) {
      is Node<T> -> this.value
      is Empty -> throw RuntimeException("Empty list")
}
```

لا تحتوي العقدة الفارغة على أي عنصر، لذا يجب أن ترمي استثناء (أو يمكننا تعريف الدالة لتكون قابلة للعَدم)، وستُرجع عقدة البيانات قيمتها فقط. لاحظ عملية التحويل الذكي للأنواع في الجانب الأيمن للوصول إلى الخاصية value، وتتبع دوال الوصول إلى البيانات الأخرى نفس الأسطر.

هنالك دالة مثيرة للاهتمام وهي الدالة append، فعندما نضيف عنصرًا إلى القائمة، ننشئ بذلك قائمة جديدة من القائمة السابقة مع إضافة العنصر إلى نهايتها في العقدة الجديدة، وللقيام بذلك، سنعرِّف الدالة التي تقبل عنصر T، وتُرجع List<T>، ومع ذلك، فإن معامل النوع في T معرَّف على أنَّه متغاير، وهذا يعني -إذا كنت تذكر- أنَّه لا يمكننا استخدام T كمعامل إدخال.

في بعض اللغات الأخرى، سيكون الحل هو إدخال معامل نوع جديد مثل U، ومن ثم استخدامه كنوع للإرجاع للقائمة الجديدة، وللقيام بذلك، نطلب أن يكون U هو النوع الوالد أو الأعلى للنوع T بحيث تكون جميع العناصر الموجودة متوافقة مع القائمة الجديدة؛ فعلى سبيل المثال، إذا كان لدينا قائمة من الأعداد الصحيحة، يمكننا إضافة عدد عشري مضاف الدقة (double) إليها ثم الحصول على قائمة من الأعداد.

لسوء الحظ، لا تدعم كوتلن هذه الوظيفة التي تجعل U نوعًا والدًا للنوع T، وهذا من شأنه أن يكون مثالًا على قيد الحد الأدنى (lower bound)، ولكن قيود الحدود العليا هي المدعومة فقط في وقت مراجعة الكتاب، ومع ذلك، يمكننا حل هذه المشكلة بطريقتين. في الطريقة الأولى، يمكننا أن نسمح باستعمال T كمعامل إدخال على الرغم من أننا أخبرنا المصرِّف في وقت سابق أنه لا يجب أن يسمح بذلك، عن طريق تجاوز (overriding) التحقق من

التباين لهذه الدالة:

```
fun append(t: @UnsafeVariance T): List<T> = when (this) {
      is Node<T> -> Node(this.value, this.next.append(t))
      is Empty -> Node(t, Empty)
}
```

لاحظ التوصيف @UnsafeVarianc، الذي يعطّل خطأ المصرِّف، وفي هذا المثال، نحن نعلم بأن هذا سيكون على ما يرام لأن تنفيذ List المكتوب يمنع من تغيير القائمة، وعلى هذا النحو، لا يمكن أن تتسبّب أخطاء عن طريق إضافة قيم غير صالحة إلى القائمة الحاليّة.

البديل الآخر هو التصريح عن الدالة append كدالة مُوسّعة، إذ يكون معامل النوع لامتغاير:

```
fun <T>List<T>.append(t: T): List<T> = when (this) {
      is Node<T> -> Node(this.value, this.next.append(t))
      is Empty -> Node(t, Empty)
}
```

يمكننا تغليف مثالنا من خلال جعل الأنواع الحقيقية خاصة، فلا يوجد سبب لكشفها، ويمكننا إضافة تابع كائن مرافق (companion object method) لنتمكن من إنشاء نسخ من List مباشرةً من خلال استعمال List نفسها، وستبدو القائمة النهائيّة كالتالي:

```
sealed class List<out T> {
      fun isEmpty() = when (this) {
            is Empty -> true
            is Node -> false
      }

      fun size(): Int= when (this) {
            is Empty -> 0
            is Node -> 1 + this.next.size()
      }
```

```
    fun tail(): List<T> = when (this) {
        is Node -> this.next
        is Empty -> this
    }

    fun head(): T = when (this) {
        is Node<T> ->this.value
        is Empty -> throw RuntimeException("Empty list")
    }

    operator fun get(pos: Int): T {
        require(pos>= 0, { "Position must be >=0" })
        return when (this) {
            is Node<T> -> if (pos == 0) head() else this.next.get(pos
- 1)
            is Empty -> throw IndexOutOfBoundsException()
        }
    }

    fun append(t: @UnsafeVarianceT): List<T> = when (this) {
        is Node<T> -> Node(this.value, this.next.append(t))
        is Empty -> Node(t, Empty)
    }

    companion object {
        operator fun <T>invoke(vararg values: T): List<T> {
            var temp: List<T> = Empty
            for (value in values) {
                temp = temp.append(value)
            }
            return temp
```

```
        }
    }
}

private class Node<out T>(val value: T, val next: List<T>) :
private object Empty : List<Nothing>()
```

وهذه أمثلة على استخدام `List`:

```
val list = List("this").append("is").append("my").append("list")

println(list.size()) // prints 4
println(list.head()) // prints "this"
println(list[1]) // prints "is"
println(list.drop(2).head()) // prints "my"
```

أنواع البيانات الجبريَّة شائعة جدًا في البرمجة الوظيفيَّة، ويمكن استخدامها لجميع طرق التجريد (abstractions). تُنفَّذ هياكل البيانات عادةً بهذه الطريقة مثل الأشجار (trees) بالإضافة إلى الهياكل التي تتبع نمط التصميم monad مثل `Either` و `Try` و `Option`، وفي الواقع، أي نوع

يطبِّق نفسه على اتحاد (union) أو نوع منتج (product type) غالبا ما يُنفَّذ بسهولة باستخدام هذا النهج.

## 11. خلاصة الفصل

قد أظهر لك هذا الفصل كيف يمكن استخدام قوة نظام الأنواع المتقدِّم في كوتلن لتحسين متانة شيفرتنا البرمجيَّة وزيادة إمكانيَّة إعادة استخدام الدوال مُعمَّمَة النوع. نظام النوع هو واحد من أكبر التحسينات التي قدمتها كوتلن زيادة عن جافا، وفي الفصول اللاحقة، سنستخدم معاملات الأنواع على أرض الواقع مع توضيح مدى فائدتها.

الفصل التاسع:

# أصناف البيانات



لقد تطرقنا إلى مصطلح صنف البيانات (data class) في الفصل الثالث، البرمجـة كائنيَّة التوجـه في كـوتلن، ومع ذلك، لم نتحدث في تفاصيل كثيرة حول ما يمكن أن تقدمه من ميزات؛ سـيغطي هـذا الفصـل عمليـة توصـيفات الأصناف والتي ستسمح لك بالحصول على شيفرة مصدريَّة خالية من التداول (boilerplate-free code).

سنتعمَّق أكثر في هذا الفصل لنعرف ماذا يفعـل المصـرِّف لنـا خلـف الكـواليس عنـد اسـتخدام صـنف بيانـات، إذ سنتعلَّم:

- ما هو التفكيك (Declarations) وكيف تصبح أصناف البيانات تلقائيًا أهلًا لعمليات التفكيك.
- كيف تحصل على تنفيذات مكتوبة للتوابع `copy`، و `toString`، و `hashCode`، و `equals`.
- قواعد يجب اتباعها عند تعريف أصناف بيانات.
- حدود أصناف البيانات.

صمِّمت أصناف البيانات للأنواع التي من المفترض أن تكون حاويات بيانات وليس أكثر، فقابليّة قراءة الشـيفرة البرمجية مهمة بالنسبة لي وعلى الأرجح لأي شخص يقرأ هذا الكتاب. عند فتح ملف مصدري، سترغب في أن تكـون قـادرًا على فهم مـا تفعلـه الشـيفرة البرمجيّـة بسـرّعة، وعنـدما يتعلّـق الأمـر بكائنـات Plain Old Java) POJO Object)، أنا متأكد من أنك ستتجنَّب كتابة الضابطات (setters) والجالبات (getters) إذا كان كـل مـا يفعلونـه هو إرجاع قيمة. وعلاوة على ذلك، فإن جسم الشيفرة البرمجيَّة للباني كبـير في كـل الحـالات، فهـو يأخـذ المعـاملات الـواردة فقـط ويسـندها إلى الحقـول المعنيـة بعـد أن يتحقـق منهـا، وهنـا يمكن لأصـناف البيانـات أن تسـاعدك، فـإذا برمجت بلغة سكالا Scala، فأنت متعوِّد على حالة باني الصنف (case class construct) وأنا متأكـد أن فكـرة وجود اختصار للسماح لبيئة البرمجية المتكاملة IntelliJ ببناء الضابطات والجالبات قد تكون بعيدة عن المثاليَّة.

يجب على المصرِّف الحديث أن يحمل عنك عبء الشفرة المتداولة. السؤال الـذي يطـرح نفسـه، لمـاذا لم تـدعم جافا هذا حتى الآن؟ هذا لا يزال لغزًا، ويمكن أن يتحقق هذا بسهولة تامـة بإضـافة توصـيف مخصصـة، والـذي يمكن للمصرِّف التعرف عليهـا وتنفيـذ المطلـوب، وبالتـالي عـدم كسـر أي شـيفرة برمجيَّة مكتوبـة. الشـيء المحـزن أنَّ هـذه الميزة لا تلوح في الأفق، لكن لحسن الحظ، لدينا كوتلن!

تخيَّل أن لدينا الصنف التالي في جافا ليمثِّل مدخلات مدوَّنة:

```java
public class BlogEntryJ {
     private final String title;
     private final String description;
     private final DateTime publishTime;
     private final Boolean approved;
     private final DateTime lastUpdated;
     private final URI url;
     private final Integer comments;
     private final List<String> tags;
     private final String email;

     public BlogEntryJ(String title, String description, DateTime 
publishTime, Boolean approved, DateTime lastUpdated, URI url, Integer 
comments, List<String> tags, String email) {
          this.title = title;
          this.description = description;
          this.publishTime = publishTime;
          this.approved = approved;
          this.lastUpdated = lastUpdated;
          this.url = url;
          this.commentCount = commentCount;
          this.tags = tags;
          this.email = email;
     }

     public String getTitle() {
          return title;
     }

     public String getDescription() {
          return description;
     }
```

```
}
```

تجاهلنا أغلب الجالبات من أجل البساطة. في هذا المثال، كل الحقول سهلة القراءة، وإذا كنت ترغب في الحصول على بنية بيانات قابلة للتغيير، فستحتاج إلى إضافة ضابطات (يجب عليك إعادة نسخة من الحقل tags إذا أردت الحفاظ على عدم قابلية التغيير، وإلا سيتمكن المستدعي من إضافة/حذف العناصر وبالتالي كسر التغليف الخاص بك). لنناقش كيف يمكنك تحقيق ذلك وأكثر كما سترى لاحقًا في كوتلن. بالنسبة إلى الشيفرة البرمجيّة في كوتلن، لقد احترت في جعل بعض الحقول قابلة للكتابة والتعديل من أجل مناقشة شيفرة البرمجيّة الخاصة بالضابطات كذلك. في مقتطف الشيفرة البرمجيّة التالي، ستلاحظ أن الحقل القابلة للكتابة عرِّف على أنَّه var، بينما عرِّف الحقل القابل للقراءة فقط على أنَّه val، وسيكون من الجميل أن يفترض المصرِّف أن نوع المتغيِّر الافتراضي هو val، فمصرِّف سكالا يفعل هذا بالفعل في حالة الأصناف:

```
data class BlogEntry(var title: String, var description: String, val
publishTime: DateTime,val approved: Boolean?, val lastUpdated: DateTime,
val url: URI, val commentCount: Int?, val topTags: List<String>, val
email: String?)

val blogEntry = BlogEntry("Data Classes are here", "Because Kotlin rulz!",
DateTime.now(), true, DateTime.now(),
URI("http://packt.com/blog/programming_kotlin/data_classes"), 0,
emptyList(), null)
```

ليس هنالك موازنة بين الإثنين، نهج كوتلن أنظف لأن جميع الشيفرات المتداولة تمت إزالتها.

قد تعتقد، في الوقت الحالي، أنك حصلت على عدد قليل من ضغطات المفاتيح، لكن هنالك الكثير من العمليات تحدث خلف الكواليس، وأنا واثق من أنك ستقدِّر هذه اللغة كثيرًا بعد معرفتها. ولرؤية جميع الأعمال التي يقوم بها مُصرِّف كوتلن من أجلنا، نحن بحاجة إلى النظر في البايتكود المولَّد.

## 1. الإنشاء التلقائي للجالبات وللضابطات

إن كان لدينا تصريح معطى لمتغيِّر من النوع var في الباني، فسينشئ المُصرِّف جالبات وضابطات بشكل تلقائي، فالمصرِّف ينشئ التابع getTitle والتابع setTitle لحقل باسم title، وهذا معناه أن التفاعل مع جافا سيترجم إلى استدعاء هذه التوابع:

```
public final java.lang.String getTitle();
     Code:
          0: aload_0
          1: getfield #11 // Field title:Ljava/lang/String;
          4: areturn
public final void setTitle(java.lang.String);
     Code:
          0: aload_1
          1: ldc #17 // String <set-?>
          3: invokestatic #23 // Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/
Object;L java/lang/String;)V
          6: aload_0
          7: aload_1
          8: putfield #11 // Field title:Ljava/lang/String;
          11: return
```

الشيفرة البرمجيّة هذه واضحة، وفي جسم الشيفرة للضابط، انظر إلى السطر 3، لدينا تحقُّق ضمني من وجود قيم العَدم عن طريق تابع المكتبة القياسي `checkParameterIsNotNull` في كوتلن. يميِّز نظام النوع بين المراجع التي يمكن أن تحتوي على قيم العَدم وتلك التي لا تستطيع. في حالة `title`، يشير تعريف النوع إلى أنَّه لا يسمح بأي قيمة عَدم، وعلى نقيض ذلك، يسمح حقل `email` بالعَدم (أي أخذ القيمة `null`)، وينعكس هذا في الشيفرة البرمجيّة المولَّدة:

```
public final void setEmail(java.lang.String);
     Code:
          0: aload_0
          1: aload_1
          2: putfield #72 // Field email:Ljava/lang/String;
          5: return
```

كما ترى، يُحذف التحقُّق الضمني لقيمة العَدم في هذه الحالة.

إذا صرَّحت عن حقل على أنه val، فسيصنع المُصرِّف التابع الجالب لقيمته فقط، وهذه هي الحال مع حقل lastUpdated، ولن أشرح البايتكود المولَّد لأنه يشبه حقل title.

## 2. التابع copy

عند استخدام صنف البيانات، ستحصل على تابع copy العجيب! يسمح لك هذا التابع بإنشاء نسخة جديدة من نوعك الخاص أثناء اختيار الحقول التي ترغب في تغييرها. فعلى سبيل المثال، قد تقرر أنك تريد الحصول على نسخة من BlogEntry وذلك من نسخة موجودة وتريد فقط تغيير الحقلين title و description:

```
blogEntry.copy(title = "Properties in Kotlin", description = "Properties
are awesome in Kotlin")
```

إذا كنت مطوّر جافا، فستلاحظ تشابهًا مع تابع clone، ومع ذلك، فإن التابع copy هو أكثر قوّة، ويسمح لك بتغيير أي من الحقول في النسخة الجديدة المنسوخة.

إذا نظرت إلى معلومات المعامل للتابع copy ‏(CTRL+ P هو الاختصار الافتراضي)، فسترى ما يلي:

```
16
17      println(blogEntry)
[title: String = ...], [description: String = ...], [publishTime: DateTime = ...], [approved: Boolean? = ...], [lastUpdated: DateTime = ...], [url: URI = ...], [comments: Int? = ...], [tags: List<String> = ...], [email: String? = ...]
19      blogEntry.copy(title = "Properties in Kotlin",
20              description = "Properties are awsome in Kotlin")
21  }
22
```

في لقطة الشاشة هذه، يمكنك أن ترى أن كل حقل موجود داخل [ ]، وهذا يعني أنه اختياري، ولفعل ذلك، يولِّد المصرِّف تابعين لنا، وهذا مقتطف من شيفرة بايتكود (مرَّة أخرى، استبعدت بعض الأسطر من أجل الوضوح):

```
public final com.programming.kotlin.chapter09.BlogEntry
copy(java.lang.String, java.lang.String, org.joda.time.DateTime,
java.lang.Boolean, org.joda.time.DateTime, java.net.URI,
java.lang.Integer, java.util.List<java.lang.String>,  java.lang.String);
Code:
0: aload_1
1: ldc           #76                 // String title
3: invokestatic  #23                 // Method
```

```
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/Ob
ject;Ljava/lang/String;)V
34: ldc           #81                 // String tags
36: invokestatic  #23                 // Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/Ob
ject;Ljava/lang/String;)V
39: new           #2        // class
com/programming/kotlin/chapter09/BlogEntry
42: dup
43: aload_1
44: aload_2
45: aload_3
46: aload         4
48: aload         5
50: aload         6
52: aload         7
54: aload         8
56: aload         9
58: invokespecial #97                 // Method
"<init>":(Ljava/lang/String;Ljava/lang/String;Lorg/joda/time/DateTime ;Lja
va/lang/Boolean;Lorg/joda/time/DateTime;Ljava/net/URI;Ljava/lang/
Integer;Ljava/util/List;Ljava/lang/String;)V
     61: areturn
public static com.programming.kotlin.chapter09.BlogEntry
 copy$default(com.programming.kotlin.chapter09.BlogEntry,
java.lang.String, java.lang.String, org.joda.time.DateTime,
java.lang.Boolean, org.joda.time.DateTime, java.net.URI,
java.lang.Integer, java.util.List, java.lang.String, int,
java.lang.Object);
Code:
 0: aload         11
2: ifnull        15
5: new           #101       // class
java/lang/UnsupportedOperationException
8: dup
9: ldc           #103       // String Super calls with  default arguments
not supported in this target, function: copy
11: invokespecial #105            // Method
```

```
java/lang/UnsupportedOperationException."<init>":(Ljava/lang/String;) V
14: athrow
15: aload_0
16: iload         10
18: iconst_1
19: iand
20: ifeq          28
23: aload_0
24: getfield      #11                 // Field  title:Ljava/lang/String;
27: astore_1
28: aload_1
29: iload         10
31: iconst_2
32: iand
33: ifeq          41
36: aload_0
37: getfield      #27                 // Field
description:Ljava/lang/String;
40: astore_2
41: aload_2
42: iload         10
44: iconst_4
45: iand
46: ifeq          54
145: aload_0
146: getfield      #72                 // Field  email:Ljava/lang/String;
149: astore        9
151: aload         9
153: invokevirtual #107                // Method
copy:(Ljava/lang/String;Ljava/lang/String;Lorg/joda/time/DateTime;Lja
va/lang/Boolean;Lorg/joda/time/DateTime;Ljava/net/URI;Ljava/lang/Inte
ger;Ljava/util/List;Ljava/lang/String;)Lcom/programming/kotlin/chapte r09/
BlogEntry;
156: areturn
```

التابع الأول المولّد هو تابع نسخة (instance method) يأخذ مجموعة من المعاملات التي تمثّل جميع

الحقول المُصرَّح عنها لصنف البيانات، وبعد جميع تحقيقات العَدم للمعاملات، تستدعي الشيفرة البرمجيَّة في السطر 58 الباني من أجل:

```
BlogEntry:58: invokespecial #97 // Method "<init>":(Ljava/lang/... .
```

الجزء الغريب هو وجود التابع الثاني، `copy$default`، وهذا التابع ساكن (static) ويأخذ نسخة `BlogEntry` كمعامل أول، متبوع بمعامل لكل حقل معرَّف، والجزء المثير للاهتمام يأتي بعد ذلك الحقل `title`. لا يُتوقَّع منك معرِّفة شيفرة البايتكود في هذا المستوى، ولكن من الجيد أن تعرف ما يحدث؛ المفتاح يكمن في السطرين: `18: iconst_1` و `28 20: ifeq`، وهذا مقتطف الشيفرة البرمجيَّة لهذا:

```
15: aload_0
16: iload          10
18: iconst_1
19: iand
20: ifeq           28
23: aload_0
24: getfield       #11                  // Field title:Ljava/lang/String;
27: astore_1
28: aload_1
```

دعني أترجم لك ما يحدث، إذا كان المعامل title مساويًا لقيمة ثابتة، فسيسترد قيمة title من النسخة، انظر للسطر `24: getfield #11`. وخلاف ذلك، فإنَّه يستخدم القيمة التي مُمرِّرت إلى التابع `copy`، قد تتساءل، كما فعلت أنا، من أي تأتي هذه الثوابت؟ السر موجود في `iconst_1` و `iconst_2` وما إلى ذلك. دعنا ننظر إلى الشيفرة البرمجيَّة المولَّدة عند استدعاء الدالة `copy`، وسيساعدنا هذا في الإجابة عن السؤال، وهذه هي شيفرة كوتلن المستخدمة:

```
fun main(args: Array<String>) {
        val blogEntry = BlogEntry("Data Classes are here","Because  Kotlin 
rulz!", DateTime.now(),true, DateTime.now(), 
URI("http://packt.com/blog/programming_kotlin/data_classes"),0, 
emptyList(),"")
        println(blogEntry)
```

```
        blogEntry.copy(title = "Properties in Kotlin",
        description = "Properties are awesome in Kotlin",
        approved = true,
        tags = listOf("tag1"))
    }
```

هذا هو البايتكود المولَّد من آخر استدعاء للتابع:

```
69: ldc #76 // String Properties in Kotlin
71: ldc #78 // String Properties are awesome in Kotlin
73: aconst_null
74: iconst_1
75: invokestatic #38 // Method
java/lang/Boolean.valueOf:(Z)Ljava/lang/Boolean;
78: aconst_null
79: aconst_null
80: aconst_null
81: ldc #80 // String tag1
83: invokestatic #84 // Method kotlin/collections/CollectionsKt.listOf:
(Ljava/lang/Object;)Ljava/util/List;
86: aconst_null
87: sipush 372
90: aconst_null

91: invokestatic  #88                  // Method
com/programming/kotlin/chapter09/BlogEntry.copy$default:(Lcom/programming/
kotlin/chapter09/BlogEntry;Ljava/lang/String;Ljava/lang/String;Lorg/joda/
time/DateTime;Ljava/lang/Boolean;Lorg/joda/time/DateTime;Ljava/net/
URI;Ljava/lang/Integer;Ljava/util/List;Ljava/lang/String;Iljava/lang/
Object;)Lcom/programming/kotlin/chapter09/BlogEntry;
```

بدءًا من السطر 69، تبدأ الشيفرة في دفع المتغيرات في المكدّس، وهي تتبع ترتيب الخاصيات المعرَّفة في صنف البيانات، وعلى سبيل المثال، نكتب فوق قيم `title` و `description` ومن ثم نقفز إلى حقل `approve`. بالنسبة لجميع القيم غير المعطاة، تكون null عن طريق برنامج البايتكود `aconst_null`. على الرغم من أنك ستستدعي التابع copy مع كائن `blogEntry`، ويستدعي بايتكود في الواقع التابع `copy$default` الساكن

وليس تابع نسخة كما يتوقع أي امرئ.

بما أن تابعًا ساكنًا عرِّف في الصنف BlogEntry، فستتوقَّع أن هذا التابع سيكون متوفرًا في القائمة المنسدلة للإكمال التلقائي (auto-completion dropdown) في بيئة التطوير، وهذا خطأ، لأن هذا التابع غير موجود أصلا.

قد تتساءل عما إذا كان يمكنك تحقيق ذلك أثناء استدعاء التابع copy من الشيفرة البرمجيَّة لجافا، حسنًا يجب أن أخيِّب ظنك، في هذه الحالة، استدعاء التابع copy سينتهي باستدعاء التابع النسخة وليس الساكن، وهذا يعني أنك لن تستفيد من الكتابة فوق مجموعة فرعيَّة من حقول النسخة.

من داخل شيفرة جافا المصدرية، أطلب من بيئة IntelliJ عرض معلومات المعامل، وستحصل على النتيجة التالية:



يجب عليك توفير جميع المعاملات عند استدعاء هذه الدالة، ويجب أن تكون غير فارغة، وستبدو الشيفرة البرمجيّة مشابهة لهذا:

```
blogEntry.copy("Properties in Kotlin","Properties are awesome in  Kotlin",
blogEntry.getPublishTime(), blogEntry.getApproved(),
blogEntry.getLastUpdated(), blogEntry.getUrl(), blogEntry.getComments(),
blogEntry.getTags(), blogEntry.getEmail());
```

## 3. التابع toString العجيب

عندما تعرِّف نوعًا جديدًا، فمن الأفضل أن تستبدل تنفيذ التابع toString فيه ليُرجع سلسلة نصيّة تصف هذا النوع. لننظر إلى الصنف BlogEntry الذي عرَّفناه في بداية الفصل، سنكتب بضعة أسطر لتوفير تنفيذٍ لهذا التابع، لكن لماذا تكتبها إذا كان بإمكانك الحصول عليها تلقائيًا وبسهولة؟ لماذا لا تدع المصرِّف يقوم بذلك نيابة عنك؟ إذا أضفت أو حذفت حقلًا جديدًا، فستتحدَّث الشيفرة البرمجيَّة تلقائيًا، وهنالك احتماليَّة لتغييرك للشيفرة البرمجيّة لجسم toString عندما تكون عملية إضافة حقل أو إعادة تسميته أو حذفه عمليةً معقَّدة:

```
public java.lang.String toString();
Code:
0: new           #122                // class  java/lang/StringBuilder
3: dup
4: invokespecial #123              // Method
java/lang/StringBuilder."<init>":()V
7: ldc           #125                // String BlogEntry(title=
9: invokevirtual #129                // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringB
uilder;
12: aload_0
13: getfield      #11                // Field  title:Ljava/lang/String;
16: invokevirtual #129               // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringB
uilder;
108: aload_0
109: getfield      #72               // Field  email:Ljava/lang/String;
112: invokevirtual #129              // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringB
uilder;
115: ldc           #150              // String )
117: invokevirtual #129              // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringB
uilder;
120: invokevirtual #152       // Method  java/lang/StringBuilder.toString:
()Ljava/lang/String;
123: areturn
```

الشيفرة البرمجيَّة سهلة الفهم، فهي تصنع نسخة جديد من النوع StringBuilder، وتضيف لكل حقل معرَّف فيه النص FIELD=VALUE (اسم الحقل وقيمته). وفي نهاية الدالة، ستُرجِع القيمة المتراكمة.

## 4. توليد التابعان hashCode و equals تلقائيًا

نعرف أنَّ كل نوع في كوتلن مشتق من النوع Any، والذي يحوي التابع hashCode، المكافئ للتابع hashCode في الصنف Object في جافا، وهذا التابع مهم عندما ترغب في وضع النسخ في تجميعات، مثل النوع Map (خريطة).

تسمح شيفرة hash[12] -التي تخص الكائنات- للخوارزميات وهياكل البيانات بوضع نسخها في دلاء (buckets)، فتخيَّل أنك قمت بكتابة تنفيذ لدفتر هاتف، فستضع أي اسم يبدأ بالحرف أ في القسم أ، وأي اسم يبدأ بالحرف ب في القسم ب، وهكذا. ويتيح لك هذا النهج البسيط الحصول على عمليات بحث أسرع عند البحث عن اسم شخص ما، وهذه هي الطريقة التي تطبَّق فيها التجميعات (collections) التي تعتمد على شيفرة hash مثل HashMap و HashSet.

عند تنفيذ التابع، تحتاج إلى الالتزام بقواعد:

1. عند استدعاء نفس الكائن أكثر من مرة خلال وقت التشغيل، يجب أن يعيد التابع hashCode نفس القيمة باستمرار، نظرًا لأن الكائن لم يُعدَّل.
2. إذا كانت نتيجة تنفيذ التابع equals على كائنين هي true، فيجب أن يرجع استدعاء التابع hashCode عليهما نفس القيمة العددية.
3. إذا كان عنصران غير متساويين، هذا يعني أن نتيجة استدعاء التابع equals عليهما تساوي false، ويجب في هذه الحالة أن يعيد تابع hashCode قيمة مختلفة لكل منهما رغم أنَّه ليس من الضروري أن يعيد هذا التابع قيمة فريدة لكل كائن، ومع ذلك، يمكن تحسين أداء التجميعات التي تعتمد على قيمة hash إذا ولدّنا عددًا صحيحًا مختلفًا للكائنات غير المتساوية.

التابع الآخر العجيب الذي سنستعمله بكثرة هو equals، ويشير إذا ما إذا كان كائنٌ مساويًا لهيكل كائن ما.

بالطبع، يمكن لبيئة التطوير IntelliJ توليد التابعين المذكورين؛ اترك جانبًا شاشة الساحر (wizard screen) التي تطلب منك تحديد الحقول والحقول التي لا تقبل قيمة عدمية، واسأل نفسك: لماذا لا يفعل المصرِّف كل هذا لنا؟ مرّة أخرى، فهذه شيفرة متداولة لن تضطر إلى تعديلها في معظم الحالات، لأن نظام نوع كوتلن يفرِّق بين الأنواع القابلة للإنعدام والغير قابلة للإنعدام، ولا نحتاج إلى المطالبة باستخدام مجموعة من حقول غير قابلة للإنعدام، إذ يملك المصرِّف كل المعلومات المطلوبة.

دعنا نولِّد توابع الصنف BlogEntryJ في جافا، في بيئة التطوير IntelliJ. اختر من قائمة Code الخيار

12 القيمة Hash هي قيمة عددية مميزة فريدة تُحسَّب لأي قيمة وفق خوارزمية معقَّدة وتُستخدّم كثيرًا في مجال التشفير والضغط ومجموع التحقق (checksum) والفهرسة.

Generate ثم اختر التابعان `equals()` و `hashCode()`. وستبدو شيفرة جافا البرمجيّة المولّدة لك مشابهة للشيفرة التالية:

```
@Override
     public boolean equals(Object o) {
        if (this == o)
           return true;
        if (o == null || getClass() != o.getClass())
           return false;

        BlogEntryJ that = (BlogEntryJ) o;
        if (!title.equals(that.title))
           return false;
        if (!description.equals(that.description))
           return false;
        if (!publishTime.equals(that.publishTime))
           return false;
        if (approved != null ? !approved.equals(that.approved) :
that.approved != null)
           return false;
        if (!lastUpdated.equals(that.lastUpdated))
           return false;
        if (!url.equals(that.url))
           return false;
        if (comments != null ? !comments.equals(that.comments) :
that.comments != null)
           return false;
        if (tags != null ? !tags.equals(that.tags) : that.tags != null)
           return false;
        return email != null ? email.equals(that.email) : that.email ==
null;
     }

@Override
     public int hashCode() {
        int result = title.hashCode();
        result = 31 * result + description.hashCode();
        result = 31 * result + publishTime.hashCode();
```

```
        result = 31 * result + (approved != null ? approved.hashCode() :
0);
        result = 31 * result + lastUpdated.hashCode();
        result = 31 * result + url.hashCode();
        result = 31 * result + (comments != null ? comments.hashCode() :
0);
        result = 31 * result + (tags != null ? tags.hashCode() : 0);
        result = 31 * result + (email != null ? email.hashCode() : 0);
        return result;
    }
```

كل هذا جيّد ومفيد، ومع ذلك، كوتلن تنتهج نهج الابتعاد عن النقرات والاختيار وتوكل المهمة للمصرِّف، والأهم من ذلك كله، لا يجب عليك توليد التابعين في كل مرة تغيِّر فيها بنية النوع إما عن طريق إعادة تسمية الحقل أو تغيير النوع الحقل أو حقل جديد بالكامل أو حذفه.

بالنسبة للقارئ المتعطش للبايتكود، هذه نسخة من الشيفرة البرمجيّة المولّدة، سنركّز على التابع hashCode فقط ويمكنك الإطلاع على equals بنفسك. ستجد أنها ستصدر نفس الشيفرة البرمجيّة لشيفرة جافا السابقة، انظر إلى السطر 16 إذ أنّ العدد الأولى 31 سيُضرب في قيمة hashCode الخاص بالحقل title الموجود في السطر 8، ومن ثم نضيف قيمة hashCode إلى حقل description، وهذه القيمة مؤقتة ومن ثم نضربها في 31 ونحصل على قيمة hashCode للحقل publishTime. والبقيّة تشبه هذا حتى حقل email، فإذا كان الحقل فارغ، فسيُترك هكذا، وانظر على سبيل المثال السطر 164 عندما يقفز إلى السطر 173 في حالة email التي تحتوي على قيمة null:

```
public int hashCode();
Code:
0: aload_0
1: getfield      #11                // Field title:Ljava/lang/String;
4: dup
5: ifnull        14
8: invokevirtual #156               // Method java/lang/Object.hashCode:
()I
11: goto          16
14: pop
```

```
15: iconst_0
16: bipush        31
18: imul
19: aload_0
20: getfield      #27                 // Field
description:Ljava/lang/String;
159: aload_0
160: getfield      #72                 // Field email:Ljava/lang/String;
163: dup
164: ifnull        173
167: invokevirtual #156                // Method
java/lang/Object.hashCode:()I
170: goto          175
173: pop
174: iconst_0
175: iadd
176: ireturn
```

## 5. التصريحات المهدومة (Destructed declarations)

إذا أنشأت نسخة من `BlogEntry` ثم ظهر لك مربع حوار الإكمال التلقائي في بيئة التطوير وتنقلت في التوابع المتاحة، فستلاحظ وجود 9 توابع، وهذه التوابع تبدأ بسلسلة من المكونات: `component1()`، و `component2()`، وهكذا حتى `component9()`؛ كل واحدة من هذه التوابع تتطابق مع واحدة من الحقول المعرَّفة بالنوع، وسيطابق نوع الإرجاع نوعَ حقل كلٍ منهما، وفي ما يلي مقتطف من `component6()` للحقل `url`:

```
public final java.net.URI component6();
  Code:
       0: aload_0
       1: getfield      #51                 // Field url:Ljava/net/URI;
       4: areturn
```

سيفكر مطورو سكالا عند قراءة هذا على الأرجح في الصنف `Product` ومطابقة النمط

(pattern matching)، وكوتلن ليست قويةً عندما يتعلّق الأمر بمطابقة النمط، ولكن لا بأس بها من هذه الناحية على أي حال.

قد تجد أنَّه من المفيد كسر الكائن إلى مجموعة من المتغيِّرات، فإذا أخذنا النسخة السابقة من `blogEntry`، فيمكننا كتابة ما يلي:

```
val (title, description, publishTime,approved, lastUpdated, url,
comments, tags, email) = blogEntry

     println("Here are the values for each
     field in the entry:
        title=$title description=$description publishTime=$publishTime
        approved=$approved lastUpdated=$lastUpdated, url=$url
comments=$comments tags=$tags email=$email")
```

إذا نفَّذت هذه الشيفرة البرمجيَّة، ستحصل على نسخة من قيمة كل حقل، ولكن كيف يعمل هذا؟ ومرّة أخرى، سيوفّر لنا البايتكود الإجابة، وإليك مقتطف للشيفرة البرمجيّة للسطر الأول في مثال الشيفرة البرمجيّة السابقة:

```
61: astore        11
63: aload         11
65: invokevirtual #66                 // Method
com/programming/kotlin/chapter09/BlogEntry.component1:()Ljava/lang/St
ring;
68: astore_2
69: aload         11
71: invokevirtual #69                 // Method
com/programming/kotlin/chapter09/BlogEntry.component2:()Ljava/lang/St
ring;
74: astore_3
117: aload         11
119: invokevirtual #93                 // Method
com/programming/kotlin/chapter09/BlogEntry.component9:()Ljava/lang/St
ring;
122: astore        10
124: aconst_null
```

كل ما فعله المصرِّف هو ترجمة الشيفرة البرمجيّة بلغة كوتلن إلى استدعاءات إلى توابع `componentN`. لن يعمل هذا النهج في الشيفرة المصدريَّة للجافا، بعد كل شيء، فهو ليس أكثر من صياغة. ومرّةً أخرى، يساعدنا المصرّف كثيرًا.

## 6. الأنواع الهادمة (Destructing types)

لابد عند التعامل مع أنواع البيانات التطرق إلى موضوع الهدم (destruction) واستعماله، لكن هل يمكننا تحقيق نفس الشيء دون صنف البيانات؟ الجواب هو نعم، كل ما عليك فعله هو توفير توابع `componentN`، إذ الشرط الوحيد هو وضع البادئة `operator` ببداية اسم التابع عند تعريفه؛ لنفترض أن لدينا الصنف `Vector3` الذي يمثِّل الإحداثيات في مساحة ثلاثيَّة الأبعاد، ومن أجل وسيط (argument)، لن نجعل هذا الصنف صنفَ بيانات:

```
class Vector3(val x:Double, val y:Double, val z:Double){
        operator fun component1()=x
        operator fun component2()=y
        operator funcomponent3()=z
     }

     for ((x,y,z) in listOf(Vector3(0.2,0.1,0.5), Vector3(-12.0, 3.145,
5.100))){
        println("Coordinates: x=$x, y=$y, z=$z")
     }
```

كما ترى، أنشأنا لكل حقل عضو التابع `componentN` المكافئ، وبسبب هذا، يمكن للمصرِّف أن يطبِّق التدمير خلال بنية حلقة `for` (for loop construct).

ماذا لو كنت تتعامل مع مكتبة أي أنَّك لا تتحكم فيها بالشيفرة المصدريَّة، لكنك ترغب في حصول على خيار تدمير النوع؟ في هذه الحالة أيضًا، يمكنك تقديم `componentN` من خلال توابع مُوسَّعة، لنفترض أنك تعمل على تطبيق في مجال إنترنت الأشياء (Internet of Things)، وتستخدم مكتبة تقدّم لك قراءة قيم المستشعرات

الخاصة بك، وإليك صنفًا معرّفًا في جافا لبيانات المستشعر:

```
public class Sensor {
        private final String id;
        private final double value;
        public Sensor(String id, double value) {
           this.id = id;this.value = value;
        }
        public String getId() {
           return id;
        }
        public double getValue() {
           return value;
        }
     }
     ...
     // شيفرة كوتلن
     operator fun Sensor.component1()= this.id
     operator fun Sensor.component2()=this.value

     for((sensorId, value) in listOf(Sensor("DS18B20", 29.2),
Sensor("DS18B21", 32.1))){
        println("Sensor $sensorId reading is $value degrees Celsius")
     }
```

إذا نفَّذت هذه الشيفرة البرمجيَّة، فستحصل على نص جميل مع قراءة قيم المستشعر، والشيفرة البرمجيّة سهلة الفهم، ويحتوي نوع جافا على حقلين معروضين عن طريق توابع الجلب. باستخدام توابع مُوسّعة في كوتلن، نحن نوفِّر التوابع المكافئة لـ componentN، وبالتالي السماح للمصرِّف باستدعائها خلال كتلة حلقة for.

## 7. قواعد تعريف صنف بيانات

إذا كانت أي من التوابع التي قدمناها للتو موجودة في صنفك بالفعل، فلن يستبدلها المصرّف بنسخته الخاصة،

ويمكنك بالتالي أن تتحكّم بشكل كامل بالمتطلبات.

عند تعريف صنف البيانات، تحتاج إلى اتباع القواعد التالية:

- يحتاج الباني الأساسي إلى معامل واحد على الأقل.
- تحتاج جميع معاملات الباني الأساسيّة إلى أن تكون `val` أو `var` حصرًا.
- لا يمكن لأصناف البيانات أن تكون `abstract` أو `open` أو `sealed` أو `inner`.
- لا يمكن لأصناف البيانات توسيع أصناف أخرى (ولكن يمكنها تنفيذ واجهات).

تتطلّب العديد من إطارات جافا صنف خاص بك توفير بانٍ افتراضي دون معاملات. تخيَّل أنك تكتب تطبيق بريد إلكتروني وتضع نموذجًا لنوع Email كالتالي (هذه طريقة مبسَّطة):

```
data class Email(var to:String = "",
                 var subject:String= "",
                 var content:String= "")
```

إن مفتاح الحصول على خيار بانٍ فارغٍ هو توفير قيمة افتراضية لكل معامل، وستتمكن بمجرَّد الحصول على هذا من كتابة `Email email = new Email();` من جافا.

إن نظرت إلى البايتكود للباني، فستلاحظ إنشاء 3 بانيات بالفعل:

```
public com.programming.kotlin.chapter09.Email(java.lang.String,
java.lang.String, java.lang.String);
  Code:
    0: aload_1
1: ldc           #36                  // String to
 3: invokestatic  #23                  // Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/Ob
ject;Ljava/lang/String;)V
6: aload_2
7: ldc           #37                  // String subject
9: invokestatic  #23                  // Method
kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/Ob
ject;Ljava/lang/String;)V
```

```
18: aload_0
19: invokespecial #41                 // Method
java/lang/Object."<init>":()V
22: aload_0
23: aload_1
24: putfield      #11                 // Field  to:Ljava/lang/String;
34: putfield      #32                 // Field  content:Ljava/lang/String;
37: return
public com.programming.kotlin.chapter09.Email(java.lang.String,
java.lang.String, java.lang.String, int,
kotlin.jvm.internal.DefaultConstructorMarker);
  Code:
0: aload_0
1: iload         4
3: iconst_1
4: iand
5: ifeq          11
8: ldc           #44                  // String
10: astore_1
11: aload_1
12: iload         4
14: iconst_2
15: iand
16: ifeq          22
19: ldc           #44                 // String
21: astore_2
22: aload_2
34: invokespecial #46                 // Method
"<init>":(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V
37: return
public com.programming.kotlin.chapter09.Email();
  Code:
     0: aload_0
     1: aconst_null
     2: aconst_null
```

```
      3: aconst_null
      7: invokespecial #52                 // Method
  "<init>":(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;ILkot lin/
  jvm/internal/DefaultConstructorMarker;)V
      10: return
```

المنطق وراء هذا يشبه منطق التابع copy، فإذا وفَّرت قيمة افتراضيّة، فإن الباني الافتراضي سيخزّن 3 قيم عَدم null، ومن ثم، سيوازن الباني الثاني المذكور مع قيم العَدم المخزَّنة، وإذا تحقَّقت هذه الحالة، فسيستخدم القيم الافتراضيَّة الموفَّرة في تصريح التابع (السطر 19).

تأتي مكتبة كوتلن القياسيَّة مع صنفين من البيانات المدعومة، وهما Pair و Triple:

```
public data class Pair<out A, out B>(public val first: A,public  val
second: B) : Serializable
```

وهذه طريقة استخدام هذه الأصناف:

```
val countriesAndCaptial = listOf( Pair("UK", "London"),  Pair("France",
"Paris"), Pair("Australia", "Canberra"))
      for ((country, capital) in countriesAndCaptial) {
         println("The capital of $country is $capital")
      }
      val colours = listOf( Triple("#ff0000", "rgb(255, 0, 0)", "hsl(0,
100%, 50%)"), Triple("#ff4000", "rgb(255, 64, 0)", "hsl(15, 100%, 50%)"))
      for((hex, rgb, hsl) in colours){
         println("hex=$hex; rgb=$rgb;hsl=$hsl")
      }
```

رغم أنَّها موجودة في المكتبة، يجب عليك أن تفضِّل دائمًا بنائها بنفسك عن طريق توفير أسماء مناسبة للأصناف، مما يجعل الشيفرة أكثر قابليّة للقراءة.

## 8. أوجه القصور

في الوقت الحالي، لا يمكنك وراثة صنف آخر عند تعريف صنف بيانات، ولتجنُّب تأخير الإصدار 1.0، قرر صانعو كوتلن وضع هذا القيد لتجنّب المشاكل التي يمكن أن يسببها هذا؛ تخيَّل صنف بيانات، Derived، يرث من صنف بيانات آخر، يدعى Base، وإذا حدث هذا، فيجب الإجابة على هذه الأسئلة:

1. هل يجب أن تتساوى نسخة من Base مع نسخة من Derived إذا كان لديهما نفس القيم لجميع الحقول المشتركة؟
2. ماذا لو قمت بنسخ نسخة من Derived من خلال مرجع من النوع Base؟

أنا متأكد أنه في المستقبل ستزال جميع القيود وأوجه القصور وسنتمكّن من كتابة شيفرة برمجيّة شبيه بالشيفرة التالية (باني Either مألوف بالنسبة لمطوري سكالا):

```
sealed abstract class Either<out L, out R> {
        data class Left<out L, out R>(val value: L) : Either<L, R>()
        data class Right<out L, out R>(val value: R) : Either<L, R>()
     }
```

## 9. خلاصة الفصل

كوتلن قوية للغاية، وأصناف البيانات فيها تثبت ذلك. لقد تعلمّت كيف تعمل اللغة والمصرِّف معًا لتزويدك ببانيات خالية من الشيفرات المتداولة، إذ نحن بحاجة إلى التخلي عن لوحة المفاتيح (لإطالة عمرها التشغيلي :-D ) مع التركيز بشكل أكثر على حل المشكلة، ولقد رأيت أيضًا كيف يمكن تدمير كائن في عدد من المتغيرات أن يكون مفيدًا للغاية، ويعزِّز الشيفرة البرمجيَّة لكي تكون أكثر قابليّة للقراءة.

في الفصل التالي، سنغطي توسيعات كوتلن (Kotlin extensions) المضافة إلى مكتبة تجميعات جافا (Java collections library)، بالإضافة إلى التطرق لموضوع الحالة القابلة للتغيير مقابل الحالة غير قابلة للتغيير، ويظهر إضافات كوتلن (Kotlin additions)، والتي تجعل استخدام التجميعات أسهل بكثير من جافا.

الفصل العاشر:

# التجميعات



يكتب معظم مطوري البرامج الكثير من الشيفرات التي تنتهي بمعالجة تجميعة من العناصر مثل القوائم (lists) والخرائط (maps) وغيرها، وإن فهم مكتبة كوتلن القياسيّة هو مفتاح لأي مطوّر كوتلن طموح، إذا كنت تعمل مع تجميعات سكالا، ستجد عددًا من أوجه التشابه، ومع ذلك، إذا كان لديك خلفية عن لغة جافا فقط، ستجد طريقة جديدة ومحسنة للتعامل مع تجميعات من الكائنات (collections of objects). ستقدّر سهولة إنجاز الكثير بكتابة القليل وهذا لا تجده إلا في كوتلن، لا تصدِّق أبالغ فقط :-).

يغطي هذا الفصل مكتبة كوتلن القياسيّة للتجميعات (collections)، وستتعلّم كيفية توسيع مكتبة تجميعات جافا لجعل عملية البرمجة اليوميّة أسهل، وسنقدّم نوعين من التجميعات: المتغيّرة وغير القابلة للتغيير، وسنتعلم كيف يتحقَّق هذا، وكيف يعمل عند التعامل مع شيفرة جافا، وسينتهي هذا الفصل بمقدّمة إلى تدفق مجرى واجهة برمجية (streaming API).

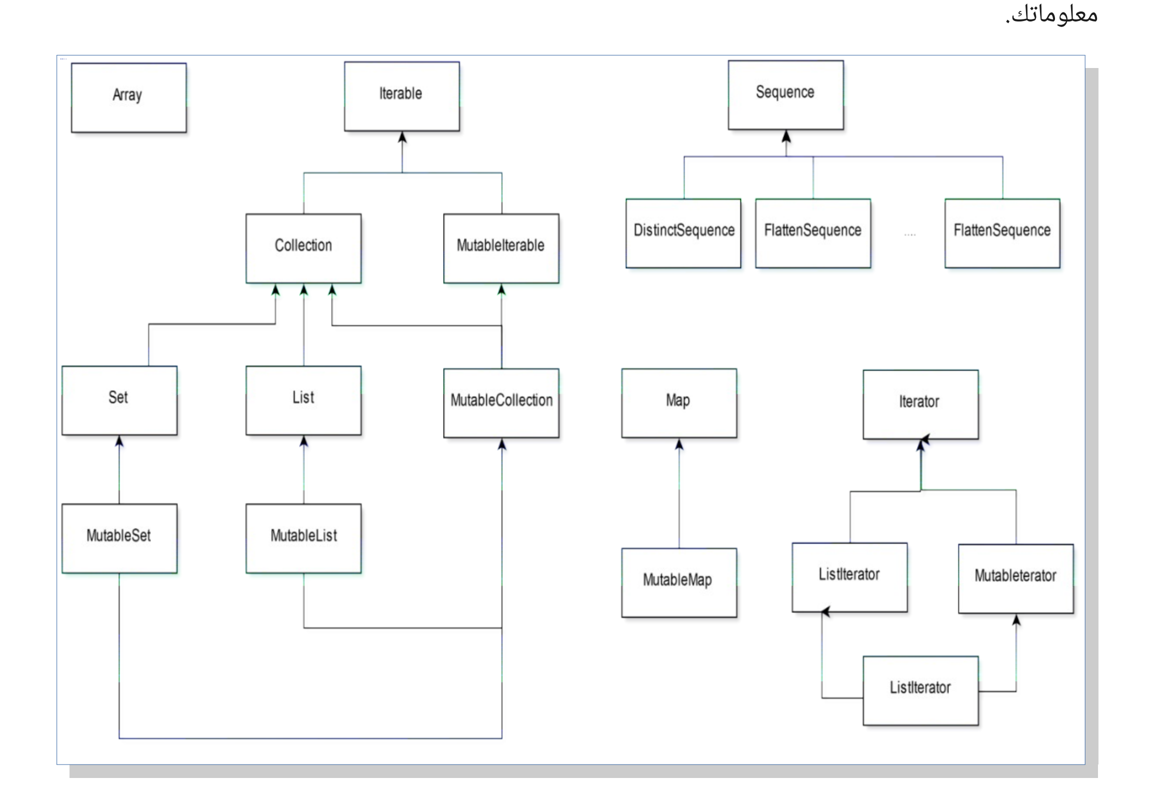
## 1. التسلسل الهرمي للصنف

مثل سكالا، تميِّز كوتلن بين التجميعات القابلة للتغيير (mutable collections) والتجميعات غير القابلة للتغيير (immutable collections). يمكن تحديث التجميعة القابلة للتغيير عن طريق إضافة عنصر، أو حذفه، أو استبداله وسينعكس هذا في حالتها؛ وعلى الجانب الآخر، توفّر التجميعة غير القابلة للتغيير، نفس العمليات (إضافة، حذف، استبدال) عن طريق إنشاء تجميعة جديدة وترك الأصليّة دون تغيير، وسنرى في وقت لاحق في هذا الفصل كيفية تحقيق عدم قابليّة التغيير عن طريق تعريف واجهة. في وقت التشغيل، تعتمد التنفيذات (implementations) على تجميعات جافا القابلة للتغيير.

على عكس سكالا، قرر صانعو كوتلن تجنُّب وجود فضاءين منفصلين لنطاق الأسماء (namespaces) لكل وضعِ تجميعةٍ (collection mode)، وستجد جميع التجميعات في فضاء الاسم `kotlin.collections`.

في الشكل التالي، سترى رسمًا تخطيطيًا لأصناف التجميعات في كوتلن، يمكن التعرّف على جميع الأنواع القابلة للتغيير بسهولة لأنها تحمل البادئة `Mutable`، وجميعها تقبل معاملات النوع وتعميم نوع معاملاتها، ومن الأشياء التي ستلاحظها ولن تجد وصفها في الرسم البياني التالي، هو أن كافة الواجهات القابلة للقراءة فقط هي متغايرة (النوع Array هو الصنف الوحيد في الرسم التخطيطي ومعامل النوع `T` هو اللامتباين الوحيد). التغاير (

(Covariant هو مصطلح يشير إلى القدرة على تغيير معامل نـوع مُعمَّم من صـنفٍ إلى صـنفٍ والـدٍ أعلى لـه، وهـذا يعني أنه يمكنك أخذ `List<String>` وتعيينها إلى `List<Any>` لأنَّ الصـنف `Any` هـو صـنفٌ والـدٌ (parent). في كـوتلن، يمكنـك الإشـارة إلى معـاملات نـوع مُعمَّم متغـاير عن طريـق الكلمـة المفتاحيـة `out` مثـل الواجهـة `Iterable<out T>`، ولقد تحدثنا عن التغاير كثيرًا بالتفصيل في الفصل الثامن ويمكنك مراجعة الفصل لمراجعة معلوماتك.




سـتجد الواجهـة `Iterable` في الجـزء العلـوي من التسلسـل الهـرمي للصـنف، وسـترى في مقتطـف الشـيفرة البرمجيّة التالية أن تعريفها بسيط:

```
public interface Iterable<out T> {
        public abstract operator fun iterator(): Iterator<T>
```

```
      }
```

تُوسّع الواجهة Collection الواجهة Iterable، وتعرِّف التوابع لتحديد وجود العناصر في المجموعة، فضلًا عن حجم المجموعة والتحقق من الحاوية الفارغة، ويمكنك التفكير في هذا التابع كمعامل استعلام لمجموعة معيَّنة:

```
public interface Collection<out E> : Iterable<E> {
        public val size: Int
        public fun isEmpty(): Boolean
        public operator fun contains(element: @UnsafeVariance E):  Boolean
        override fun iterator(): Iterator<E>
        public fun containsAll(elements: Collection<@UnsafeVariance E>):
Boolean
      }
```

الواجهة MutableIterable هي أخت الواجهة Collection وأوجدت لإعادة تعريف التابع iterator() الأب لها لإرجاع مُكرِّر قابل للتغيير (mutable iterator) بدلًا من واحد غير قابل للتغيير:

```
public interface MutableIterable<out T> : Iterable<T> {
       override fun iterator(): MutableIterator<T>
      }
```

ومن يُشتَق من الصنف Collection النوع الأكثر استخدامًا وهو الصنف List (قائمة) والتي هي مجموعة مرتَّبة من العناصر، وتدعم توابع هذا الصنف وصولَ قراءةٍ فقط إلى التجميعة. وأكثر دالة بارزة هي get، وهي تسمح بجلب عنصر وفقًا لفهرس موضعه:

```
public interface List<out E> : Collection<E> {
        // Query Operations
        override val size: Int
        override fun isEmpty(): Boolean
        override fun contains(element:
        override fun iterator(): Iterator<E>
```

```
        override fun containsAll(elements: Collection<@UnsafeVariance
E>): Boolean
        public operator fun get(index: Int): E
        public fun indexOf(element: @UnsafeVariance E): Int
        public fun lastIndexOf(element: @UnsafeVariance E): Int
        // List Iterators
        public fun listIterator(): ListIterator<E>
        public fun listIterator(index: Int): ListIterator<E>
        // View
        public fun subList(fromIndex: Int, toIndex: Int): List<E>
    }
```

الواجهة التالية المشتقة من `Collection` هي `Set` (طقم)، والتي هي تجميعة عناصر فريدة وغير مرتَّبة، وتدعم الدوال في هذه الواجهة وصول قراءةٍ فقط:

```
public interface Set<out E> : Collection<E> {
        // Query Operations
        override val size: Int
        override fun isEmpty(): Boolean
        override fun contains(element: @UnsafeVariance E): Boolean
        override fun iterator(): Iterator<E>
        // Bulk Operations
        override fun containsAll(elements: Collection<@UnsafeVariance
E>): Boolean
    }
```

لم نشاهد حتى الآن سوى أنواع التجميعات القابلة للقراءة فقط/غير القابلة للتغيير، والتجميعات التي تسمح بتعديل عناصرها مدعوم عبر الواجهة `MutableCollection`، ويعرض مقتطف الشيفرة البرمجيّة التالية جميع التوابع المعرَّفة عن طريق هذه الواجهة:

```
public interface MutableCollection<E> : Collection<E>,  MutableIterable<E>
```

```
{
        // Query Operations
        override fun iterator(): MutableIterator<E>

        // Modification Operations
        public fun add(element: E): Boolean
        public fun remove(element: E): Boolean

        // Bulk Modification Operations
        public fun addAll(elements: Collection<E>): Boolean
        public fun removeAll(elements: Collection<E>): Boolean
        public fun retainAll(elements: Collection<E>): Boolean
        public fun clear(): Unit
     }
```

تعد الواجهة `MutableCollection` أكثر تخصصًا عن طريق `MutableList` إذ توسّع الأخيرة توابع الأولى بإضافة أخرى إليها تسمح باستبدال أو حذف عنصر بناءً على فهرس موضعه:

```
public interface MutableList<E> : List<E>, MutableCollection<E> {
        // Modification Operations
        override fun add(element: E): Boolean
        override fun remove(element: E): Boolean

        // Bulk Modification Operations
        override fun addAll(elements: Collection<E>): Boolean
        public fun addAll(index: Int, elements: Collection<E>): Boolean
        override fun removeAll(elements: Collection<E>): Boolean
        override fun retainAll(elements: Collection<E>): Boolean
        override fun clear(): Unit

        // Positional Access Operations
        public operator fun set(index: Int, element: E): E
```

```
        public fun add(index: Int, element: E): Unit
        public fun removeAt(index: Int): E

        // List Iterators
        override fun listIterator(): MutableListIterator<E>
        override fun listIterator(index: Int): MutableListIterator<E>

        // View
        override fun subList(fromIndex: Int, toIndex: Int):
MutableList<E>
     }
```

وبالمثـــل، كمـــا أنَّ لـــدينا الطقم Set الغـــير قابـــل للتغيـــير فلـــدينا مكافئـــه القابـــل للتغيـــير، وهي الواجهة MutableSet:

```
public interface MutableSet<E> : Set<E>, MutableCollection<E> {
        // Query Operations
        override fun iterator(): MutableIterator<E>

        // Modification Operations
        override fun add(element: E): Boolean
        override fun remove(element: E): Boolean

        // Bulk Modification Operations
        override fun addAll(elements: Collection<E>): Boolean
        override fun removeAll(elements: Collection<E>): Boolean
        override fun retainAll(elements: Collection<E>): Boolean
        override fun clear(): Unit
     }
```

ستلاحظ أن كل من Map و MutableMap لا يرثان من أي من الواجهات الـتي ناقشـناها سـابقًا، وقـد تتسـاءل، كيف يمكننا التكرار (iterate) عليهما؟ إذا كنت تتذكّر في الفصل التاسع، أصناف البيانات، تناقشـنا حـول تـدمير Ma

p، ولقد ذكرنا تابعين للتكرار وهما component1 و component2، ولذلك يمكننا التكرار على خريطة Map بفضل تابع مُوسّع يدعى iterator، فالخريطة Map هي عبارة على تجميعة تُخزِّن أزواجًا من الكائنات، كل زوج يمثِّل مفتاحًا وقيمة، وتدعم استرجاع القيمة المرتبطة بمفتاحها بطريقة فعّالة.

مفاتيح الخريطة Map فريدة ويمكنها تخزين قيمة واحدة فقط لكل مفتاح، وتوفِّر التوابع المعرّفة في الواجهة Map عملية القراءة فقط:

```
public interface Map<K, out V> {
        //Query Operations
        public val size: Int
        public fun isEmpty(): Boolean
        public fun containsKey(key: K): Boolean
        public fun containsValue(value: @UnsafeVariance V): Boolean
        public operator fun get(key: K): V?


        public fun getOrDefault(key: K, defaultValue: @UnsafeVariance  V):
V {
          //See default implementation in JDK sources
          return null as V
        }

        //Views
        public val keys: Set<K>
        public val values: Collection<V>
        public val entries: Set<Map.Entry<K, V>>
        public interface Entry<out K, out V> {
          public val key: K
          public val value: V
          }
     }
```

من أجل دعم قابلية التغيير، أضيف إلى التسلسل الهرمي النوع MutableMap، فستجد في الشيفرة البرمجيّة التالية تعريفه وستجد أيضًا التوابع remove أو put أو putAll أو clear:

```
public interface MutableMap<K, V> : Map<K, V> {
    //Modification Operations
    public fun put(key: K, value: V): V?
    public fun remove(key: K): V?
    //Bulk Modification Operations
    public fun putAll(from: Map<out K, V>): Unit
    public fun clear(): Unit

    //Views
    override val keys: MutableSet<K>
    override val values: MutableCollection<V>
    override val entries: MutableSet<MutableMap.MutableEntry<K, V>>
    public interface MutableEntry<K,V>: Map.Entry<K, V> {
        public fun setValue(newValue: V): V
    }
}
```

يبقى الصنف Array وحيدًا في الرسم البياني الذي لم نتحدث عنه، فالمصفوفة -أي النوع Array- هي مجـرَّد حاوية لعدد محدَّد من القيم ذات نوع واحد، إذ يحدَّد نوع المصفوفة وقت إنشائها ولا يمكن تغييره بعدئذ:

```
public class Array<T> : Cloneable {
    public inline constructor(size: Int, init: (Int) ->T)
    public operator fun get(index: Int): T
    public operator fun set(index: Int, value: T): Unit
    public val size: Int
    public operator fun iterator(): Iterator<T>
    public override fun clone(): Array<T>
}
```

على الجانب الأيسر السفلي من الرسم البياني، يمكنك ملاحظة مجموعة المُكرِّرات، فالمُكرِّر (iterator) على تجميعة (collection) يمكن أن يمثِّل على أنَّه تسلسلًا من العناصر، توفِّر كوتلن الدعم للمُكرِّرات القابلة للتغيير والغير القابلة للتغيير، ولذلك سيُرجع كل نوع collection تنفيذ المُكرِّر المقابل، فعلى سبيل المثال، ستُرجع قائمة تنفيذ المُكرِّرات، في حين أنَّ MutableList ستُرجع نسخة من MutableIterator.

```
public interface Iterator<out T> {
        public operator fun next(): T
        public operator fun hasNext(): Boolean
    }
public interface MutableIterator<out T> : Iterator<T> {
        public fun remove(): Unit
    }
```

عادةً يقرأ المُكرِّر Iterator باتجاه الأمام فقط (forward reading)، وهذا يعني أنَّك لا تستطيع العودة إلى العنصر الذي مررت عليه سابقًا، ولدعم هذه الوظيفة، تحتوي المكتبة على ListIterator، وبالتالي يمكن للمستدعي المُرور للأمام والخلف على المجموعة الأساسيَّة. ويوجد نوعين لهذا: القابل للتغيير والغير قابل للتغيير، فالنسخة القابلة للتغيير تسمح بإضافة عناصر، أو إزالتها، أو استبدالها أثناء انتقالك للتجميعة الضمنية:

```
public interface ListIterator<out T> : Iterator<T> {
        // Query Operations
        override fun next(): T
        override fun hasNext(): Boolean
        public fun hasPrevious(): Boolean
        public fun previous(): T
        public fun nextIndex(): Int
        public fun previousIndex(): Int
    }

    public interface MutableListIterator<T> : ListIterator<T>,
MutableIterator<T> {
        // Query Operations
```

```
        override fun next(): T
        override fun hasNext(): Boolean

        // Modification Operations
        override fun remove(): Unit
        public fun set(element: T): Unit
        public fun add(element: T): Unit
    }
```

ترتبـط المجموعـة (group) المتبقيـة من الواجهـات بالتسلسـلات (انظـر الجـانـب الأيمن العلـوي من الرسـم البياني)، إذ تُرجع السلسلة قيمًا من خلال مُكرِّر iterator، فجميـع قيم السلسـلة مقيَّمـة تقييمًا كسـولًا، ويمكن أن تكون هذه السلسلة لا نهائية. يمكن المُرور على معظم السلاسل عدَّة مرات، ولكن هنالك بعض التطبيقات الـتي تقيِّدك إلى المُرور مرَّة واحدة فقط عليها، فتسطيح السلسلة (flattening sequence) هو واحد من تلك الاسـتثناءات. وتحتوي الواجهة Sequence على تابع واحد فقط:

```
public interface Sequence<out T> {
        public operator fun iterator(): Iterator<T>
    }
```

يمكن ترجمة جميع التجميعات المعروضة سابقًا إلى سلسلة عبر التابع المُوسّع asSequence، وتوفّر المُكرِّرات (iterables) والمصفوفات التنفيذ الخاص بهما كما سنرى في وقت لاحق.

واحد من أهم الأشياء التي يجب فهمها هو أن كوتلن لا توفِّر تنفيذًا خاصًا بها لأنواع التجميعـات، ولكن بـدلًا من ذلك، تُستعمَل تجميعات جافـا الموجـودة، وإذا أردت البحث على الشـيفرة المصـدرية لكوتلن لتطـبيق الواجهـة List، على سبيل المثال، فستضـيِّع وقتـك، فهي غـير موجـودة، فالسـحر يحـدث وقت التصـريف. تتعامـل كـوتلن مـع بعض أصناف مجموعة جافا بطريقة خاصة: فهي تربط نوع جافا إلى نوع كوتلن، ولا يتوسَّع هـذا الارتبـاط وقت التشـغيل، وستبقى أنواع جافا دون تغيير وقت التشغيل، وفي ما يلي جـدولٌ يوضِّـح بالتفصـيل التعـيين بين أنـواع تجميعـات جافا ومقابلها الغير قابل للتغيير والقابل للتغيير في كوتلن:

| نوع جافا | نوع كوتلن الثابت | نوع كوتلن القابل للتغيير | نوع المنصة |
|---|---|---|---|
| Iterator<T> | Iterator<T> | MutableIterator<T> | (Mutable) Iterator<T>! |
| Iterable<T> | Iterable<T> | MutableIterable<T> | (Mutable) Iterable<T>! |
| Collection<T> | Collection<T> | MutableCollection<T> | (Mutable) Collection<T>! |
| Set<T> | Set<T> | MutableSet<T> | (Mutable) Set<T>! |
| List<T> | List<T> | MutableList<T> | (Mutable) List<T>! |
| ListIterator<T> | ListIterator<T>l | MutableListIterator<T> | (Mutable) ListIterator<T>! |
| Map<K, V> | Map<K, V> | MutableMap<K, V> | (Mutable) Map<K, V>! |

كوتلن هي لغة آمن من القيم الفارغة المعدومة حسب تصميمها، وبسبب التشغيل المتداخل لجافا، كان على فريق كوتلن أن يريحوا نظام الأنواع قليلًا، ولذلك، قدّموا مصطلح «نوع المنصة» (platform type)، فنوع النظام الأساسي ليس سوى نوع قادم من منصة JVM الأساسيَّة، وسوف يحصل على معاملة خاصة:

- لن يفرض مصرِّف كوتلن الأمان من القيم الفارغة للمتغيّرات القادمة من جافا، ولذلك فيمكن أن يُرمَى الاستثناء NullPointerExcepiont لها.
- لا يمكنك تسمية أنواع المنصة في شيفرة كوتلن الخاصة بك، لكن سترى أن بيئة IntelliJ تعرضها مع علامة تعجب في النهاية مثل String! و ArrayList<Int!>!.
- عند تخزين نوع المنصة، فسيتوجب عليك اختيار نوع كوتلن، وستفعِّل كوتلن ذلك نيابة عنك، ولكن يمكنك ضبطها، فلنفترض أن لديك شيفرة جافا التالية: String getName() فيمكنك كتابة هذا في كوتلن:

- ◦ `val name=getName` (ستعرض بيئة التطوير المتكاملة `String!` كنوع) أو
- ◦ `val name:String?= getName()` أو
- ◦ `name:String = getName()`

- كما في النقطة السابقة، عند إعادة كتابة تابع معرَّف في جافا، فستحتاج إلى تحديد نـوع كـوتلن؛ لنفـترض أن لدينا تابعًا في جافا معرَّف بالشكل `void addFlag(String flag)`، فإذا أردت إعادة كتابة هذا التابع في كوتلن، فستحتاج إلى اختيار واحد من الخيارات التالية:
  - ◦ `override fun addFlag(flag:String):Unit` أو
  - ◦ `override fun addFlag(flag:String?)`.

يسمح حدوث ربط النوع هذا في تحديد نمط التصريف (compile type) مما يسمح بتصريف هذه الشيفرة البرمجيَّة وعملها:

```
fun <T> itWorks(list: List<T>): Unit {
        println("Java Class Type:${list.javaClass.canonicalName}")
      }

      val jlist = ArrayList<String>()
      jlist.add("sample")
      itWorks(jlist)
      itWorks(Collections.singletonList(1))
```

تصرِّح هـذه الشـيفرة عن تـابع يأخـذ المعامـل `list` لكـوتلن، ومن ثم تسـتدعيه مـرتين، ممـا تـوفر معـاملين مختلفين: `java.util.ArrayList` و `java.util.Collections.SingletonList`، ففي الحالـــة الأولى، سيفسّـر المصرِّف النـوع على أنَّه `List<String>` وإذا حـرَّكت الفـأرة فـوق `singletonList` في بيئـة التطوير فستجد تلميحًا يعرض نوع المنصة، `(Mutable)List<Int!>!`.

## 2. المصفوفات (النوع Array)

لقد تناولنا بالفعل ما هي المصفوفة (array) في القسم السابق ولقد حـان الـوقت الآن لإلقـاء نظـرة على كيفيَّة

العمل مع المصفوفات بتفاصيل أكثر قليلا.

يمكن التصريح عن مصفوفة وتهيئتها على النحو التالي:

```
val intArray = arrayOf(1, 2, 3, 4)
     println("Int array:${intArray.joinToString(",")}")
     println("Element at index 1 is:${intArray[1]}")

     val stringArray = kotlin.arrayOfNulls<String>(3)
     stringArray[0] = "a"
     stringArray[1] = "b"
     stringArray[2] = "c"
     // stringArrays[3]="d" --throws index out of bounds exception
     println("String array:${stringArray.joinToString(",")}")

     val studentArray = Array<Student>(2) { index ->
       when (index) {
         0 -> Student(1, "Alexandra", "Brook")
         1 -> Student(2, "James", "Smith")
         else ->throw IllegalArgumentException("Too many")
       }
     }
     println("Student array:${studentArray.joinToString(",")}")
     println("Student at index 0:${studentArray[0]}")

     val longArray = emptyArray<Long>()
     println("Long array:${longArray.joinToString(",")}")
```

يمكنك أن ترى هنا أربعة طرائق لتهيئة مصفوفة خاصة بك؛ النهج الأول هو الاستفادة من التابع `arrayOf` لتهيئة مصفوفة من أعداد صحيحة كما رأينا في المثال السابق.

النهج الثاني هو استخدام التابع `arrayOfNulls` لإنشاء مصفوفة ذات حجم محدَّد وتُسنَد القيمة المعدومة null لجميع عناصرها، ويمكن جلب قيمة أحد عناصرها بالشكل `studentArray[0]`.

الطريقة الثالثة تستخدم نهج باني صنف مصفوفة، الذي هو `Array`، إذ يُمرَّر إليه حجم المصفوفة ودالة لامدا (lambda) تُهيِّئ كل عنصر من عناصرها.

الطريقة الرابعة موضَّحة عمليًا في المثال الأخير، عندما استعملنا `emptyArray`، الذي أنشأنا فيه مصفوفة فارغة بطريقة اصطلاحيَّة.

تحصل المصفوفات في JVM على معاملة خاصة جدًا، لذلك يجب أن ينتهي الأمر بمصفوفة كوتلن مترجمة إلى بايتكود مشابه، وبخلاف ذلك ستتخرَّب التوافقيَّة. سنحصل على الإجابات بالنظر إلى بايتكود المولَّد التالي:

```
Compiled from "ArraysCollection.kt"
public final class  com.programming.kotlin.chapter10.ArraysCollectionKt {
   public static final void arrays();
      Code:
         0: iconst_4
   1: anewarray     #8                   // class java/lang/Integer
   4: dup
   5: iconst_0
   6: iconst_1
   7: invokestatic  #12  // Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   10: aastore
   35: checkcast     #14                  // class "[Ljava/lang/Object;"
   38: checkcast     #16                  // class "[Ljava/lang/Integer;"
   41: astore_0
```

يكمن الجواب في تعليمات `anewarray`، فتنص تعليمات بايتكود على ما يلي: `anewarray <type>`، حيث أن `<type>` هو اسم الصنف أو الواجهة، مثل `java/lang/String`، ويحجز برنامج (routine) بايتكود مصفوفةً جديدةً لتخزِّن مراجع لكائن، تنتج عددًا صحيحًا يمثِّل حجم المصفوفة، وعند استخدام هذا، سيبني مصفوفة جديدة لاحتواء مراجع للنوع المشار إليه بـ `<type>`:

```
anewarray     #8            // class
com/programming/kotlin/chapter10/Student
```

يضاف مرجعٌ للمصفوفة الجديدة إلى المكدّس عن طريق التعليمة astore_0 على مستوى البايتكود، والشيء الغريب هو أنه لا يمكنك رؤية تعقُّب (trace) للصنف Array الحالي وبانيه، وقد تسأل نفسك ما الذي يجري؟ وسأخبرك عن ذلك بعد قليل.

توفّر مكتبة كوتلن القياسيّة دعمًا مميزًا لمصفوفات من الأنواع الأساسية: intArrayOf و longArrayOf و charArrayOf و doubleArrayOf وهكذا، وستحصل لكل واحدة منها على النسخة المقابلة من صنف كوتلن: IntArray و LongArray و CharArray

و DoubleArray، والجزء المهم والمثير للاهتمام هو عدم ارتباط أيٍّ من تلك الأصناف بالنوع Array المقدّم سابقًا ولا تعدُّ أنَّها مشتقة منه حتى. لننظر إلى مثال عن إنشاء مصفوفة لنوع أساسي وليكن نوع الأعداد الصحيحة:

```
val ints = intArrayOf(1,2,3, 4, 5, 6, 7, 8, 9, 10)
println("Built in int array:${ints.joinToString(",")}")
```

هذه المرّة، سيكون النوع IntArray خلافًا للنوع Array<Int> كما عرَّفنا ذلك سابقًا. وبالنظر إلى بايتكود المولَّد، سنرى أن النوع قد تغيَّر موازنةً بالمثال السابق:

```
0: iconst_3
1: newarray       int
3: dup
4: iconst_0
5: bipush        100
7: iastore
20: astore_0
```

تغيَّر برنامج البايتكود المستخدم في إنشاء المصفوفة، فهذه المرَّة، استُخدمَت التعليمة newarray، إذ تُستخدم هذه لتخصيص مصفوفات أحاديَّة البعد من الأنواع الأساسية: boolean أو char أو float أو double أو byte أو short أو int أو long خلافًا للتعليمة anewarray، وأضيف هذا التحسين على مستوى JVM لتجنُّب عمليات التعليب (boxing) والإخراج من العلبة (unboxing)، وذلك لرفع الأداء.

بأخذ ذلك بالحسبان، يجب عليك التأكد من أنَّك تستخدم دائمًا ***ArrayOf عند تعاملك مع الأنواع الأساسية (primitive types) بدلًا من arrayOf، وإذا لم تفعل ذلك، فسيدفع برنامجك ثمن ذلك بانخفاض

الأداء المرتبط بعمليات التعليب والإخراج من العلبة.

يكمن السبب في أننا لا نرى أي تعقُّب (trace) للصنف `Array` لكوتلن في البايتكود المولَّد هو لأنها اسم مستعار لنوع جافا، وتحدث عملية ربط النوع هذه وقت التصريف، ولأسباب تتعلّق بالأداء، تصرَّف مباشرةً إلى مصفوفات جافا، ويرجى تذكُّر أن `int[]` Java يُربَط إلى `IntArray` (وهذا صالح للأنواع الأساسية المذكورة)، ويُربَط `String[]` أو `T[]` إلى `Array<out String/T>`.

يعطي المصرِّف المصفوفات معاملة خاصة، فعند التصريف إلى بايتكود JVM، سيحسِّن البايتكود المُولَّد لتجنُّب أي حِمل زائد على الأداء:

```
val countries = arrayOf("UK", "Germany", "Italy")
for (country in countries) {
  print("$country;")
}
```

من المحتمل أنَّك قد توقعت استعمال `for` للاستفادة من المُكرِّر والمُرور على كل عنصر من عناصر المصفوفة، ومع ذلك، ليس هذا هو الحال، فلا يُستخدم أي مُكرِّر:

```
val numbers = intArrayOf(10, 20, 30)
for (i in numbers.indices) {
  numbers[i] *= 10
}
```

يُطبَّق المبدأ نفسه أثناء التكرار على فهرس عبر المصفوفة، وبالإضافة إلى ذلك، لا تستفيد عملية جلب قيمة أو إسناد أخرى في المصفوفة من التابعان `get` و `set` المتوافران فيها، ومرّة أخرى، وذلك لتحسين الأداء.

تحسين آخر يستخدمه المصرِّف عندما تكون لديك كتلة `if` كالتالي:

```
val index=Random().nextInt(10)
      if (index in numbers.indices) {
         numbers[index]=index
      }
```

في هذه الحالة، تصرَّف تعليمة if كما لو كُتبت كالتالي:

```
if (index >=0 && index < numbers.size) {}
```

تأتي قوّة المكتبة القياسيّة من خلال إثراء واجهة برمجة التطبيقات مما يسمح لنا بالتعامل مع المصفوفات. في مكتبة كوتلن القياسية، يوجد ضمن الصنف kotlin.collections صنفٌ فرعي يسمى ArraysKt، وستجد ضمنه العديد من الدوال المساعدة (توابع مُوسّعة)، تشملها Array<T> ومصفوفات الأنواع الأساسية IntArray و FloatArray و ByteArray وغيرها، ولن نمر على كل واحدة، لكن سنتطرق إلى بعضها واطلع على البقية بنفسك:

```
println("First element in the IntArray:${ints.first()}")
     println("Last element in the IntArray:${ints.last()}")
     println("Take first 3 elements of the  IntArray:$
{ints.take(3).joinToString(",")}")
     println("Take last 3 elements of the IntArray:$
{ints.takeLast(3).joinToString(",")}")
     println("Take last 3 elements of the IntArray:$
{ints.takeLast(3).joinToString(",")}")
     println("Take elements smaller than 5 of the  IntArray:${
        ints.takeWhile {
           it <5
        }
        .joinToString(",")
     }")
     println("Take every 3rd element in IntArray: ${ints.filterIndexed  {
        index, element -> index % 3 == 0
     }
     .joinToString(",")}")
```

لنطّلع على كل مثال من الأمثلة السابقة التي ذكرناها. نبدأ بالتابع first() الذي هو تابع مُوسَّع هو يُرجع أوَّل عنصر في التجميعة، إذ يشرح اسمه عمله، ويشبه الاستدعاء ints[0] تقريبًا، وسبب قولي تقريبًا لأنه في حالة مصفوفة فارغة، فستحصل على الاستثناء NoSuchElementException مقابل

IndexOutOfBoundsException في ذلك الاستدعاء. نفّذ الشيفرة البرمجيّة السابقة وسترى حواية المخرجات على الرقم 1.

يستخدم المثال التالي التابع ()last لجلب آخر عنصر من التجميعة، كما توقعت من اسمه، إذ فهم أسماء التوابع شيء مهم للغاية، وهذه العمليّة سريعة، لأنّها تأخذ حجم المصفوفة، وتطرح 1 منها وتستخدم عامل get لجلب العنصر الأخير.

ننتقل إلى التابع المُوسّع take(n) الذي يرجع أول n عنصر من التجميعة التي استدعي معها، وتستخدمه جميع العمليات التي تعيد تجميعة فرعيّة من تجميعة أساسية، لكن الجزء المثير للاهتمام هو أنّ نوع الإرجاع ليس IntArray، لكن <List<Int. يمكنك أن ترى في مقتطف الشيفرة البرمجيّة التالية أن التطبيق الفعلي يعتمد على تنفيذ جافا للدالة ArrayList:

```
public fun IntArray.take(n: Int): List<Int> {
        require(n >= 0) { "Requested element count $n is less than 
zero." }
        if (n == 0) return emptyList()
        if (n >= size) return toList()
        if (n == 1) return listOf(this[0])
        var count = 0
        val list = ArrayList<Int>(n)
        for (item in this) {
           if (count++ == n)
              break;
           list.add(item)
        }
        return list
     }
```

السطر التالي من الشيفرة البرمجيّة يقوم بنفس الشيء: يأخذ ثلاثة عناصر من المصفوفة، ليس من البداية، لكن من نهاية المجموعة، وعند تشغيل هذه الشيفرة البرمجية سيطبع الأعداد 8 و 9 و 10 على الطرفية.

هنالك سيناريوهات ترغب فيها بإرجاع عناصر من مصفوفة عند بناءً على تحقّق شرط معيّن، فستجد عند قراءة توثيق أي دالة وكيفية استعمالها الاسم predicate وهي دالةٌ تتحقَّق مما إذا كان العنصر أصغر من 5، في المثال الذي وفرناه آنفًا، وستكون مخرجات هذا المثال هي أعداد من 1 إلى 4.

يمكننا استعمال التابع المُوسّع filterIndexed الذي يرجع كل س عنصر (كل عنصر ثالث في المثال القبل السابق) إذ يأخذ دالة لامدا مع معاملين، الأول هو موضع العنصر الحالي في المصفوفة والثاني هو العنصر الفعلي، مخرجات ذلك المثال ستطبع الأعداد 1 و 4 و 7 و 10.

التابعان المُوسّعان المفضلان لي هما map و flatMap، وهما مألوفان لأي مطوّر Scala، لكن تذكّر أن كوتلن ليست لغة وظيفيّة، وبالتالي، فإن مفهوم monad غير قابل للتطبيق. لن أفسّر هذا المفهوم لأنَّه يتجاوز الغرض من هذا الكتاب، وبدلًا عن ذلك، أحثك على القراءة حول هذا الموضوع، حتى إذا لم تكن تفكّر حتى في الانتقال إلى لغة وظيفيّة.

تسمح لك الدالة map بترجمة نوع العنصر الأساسي إلى عنصر مختلف إذا كان لديك مثل هذه المتطلبات، وأبسط مثال على ذلك هو ترجمة IntArray إلى تجميعة من سلاسل نصيّة:

```
val strings = ints.map { element ->"Item " + element.toString() }
     println("Transform each element IntArray into a  string:$
{strings.joinToString(",")}")
```

إذا شغَّلت هذه الشيفرة البرمجيّة، فسترى المخرجات التاليّة:

```
Transform each...: Item 1, Item 2, Item 3,..., Item 10
```

دعنا نرى كيف تُنفَّذ هذا؛ يحتوي المقتطف التالي على الشيفرة البرمجيّة للمكتبة القياسيّة الفعليّة:

```
public inline fun <R> IntArray.map(transform: (Int) ->R): List<R> {
        return mapTo(ArrayList<R>(size), transform)
     }


     public inline fun <R, C : MutableCollection<in R>>
IntArray.mapTo(destination: C, transform: (Int) ->R): C {
```

```
        for (item in this)
          destination.add(transform(item))
        return destination
      }
```

يُعـــــــرَّف نفس التـــــــابع المُوسَّـــــــع map من أجـــــــل LongArrays و DoubleArrays و ByteArray وغيرهما بالإضافة إلى الصنف Array<T>. ويمكنك بهذه الطريقة العمل مع تجميعات واجهة برمجة التطبيقات (API) بطريقة موحَّدة على الرغم من عدم وجود أي نوع من العلاقات، فكل ما يقوم به التابع map هو توجيه الاستدعاء إلى تابع توسيع آخر، mapTo، أثناء تمرير ArrayList لجافا كأول معامل وتعبير لامدا كمعاملٍ ثانٍ، ويُكرِّر التابع mapTo تطبيق تابع التحويل على كل عنصر من عناصر التجميعة المستهدفة، وتضاف نتيجة كل تحويل لكل عنصر إلى العنصر المقابل من التجميعة الهدف، وفي هذه الحالة هي ArrayList لجافا.

قد لا تزال تتساءل كيف يعمل mapTo عند توفير النوع ArrayList الخاص بجافا؛ وبعد كل شيء، يطلب التابع المُوسَّع من المجموعة الهدف أن ترث من MutableCollection. كما تناقشنا سابقًا، ليس هنالك سحر في هذا، فأثناء التصريف، يصبح اسم نوع تجميعة جافا اسمًا مستعارًا لنوع تجميعة كوتلن، وفي هذه الحالة، سيُعامِل المصرّف مرجع ArrayList كنسخة MutableCollection الخاصة بكوتلن، ولكن لا تزال تتعامل مع مجموعة جافا وقت التشغيل.

يُرجع التابع المُوسَّع flatMap قائمة مدمجة بكل التجميعات التي أُرجعت بواسطة عملية التحويل لامدا المُطبَّقة، ومن المتوقع في هذه الحالة أن يكون نوع إرجاع دالة لامدا هو Iterable<T>. وبعبارة أخرى، تسطّح flatMap سلسلةً من نُسخ Iterable، وإليك مثال نضاعف فيه كل عنصر من المصفوفة إلى ثلاثة أضعاف:

```
val charArray = charArrayOf('a', 'b', 'c')
     val tripleCharArray = charArray.flatMap { c ->charArrayOf(c, c, c).asIterable() }
     println("Triple each element in the  charArray:${tripleCharArray.joinToString(",")}}")
```

النتيجة هي قائمة (نعم، غيَّرنا نوع الحاوية) مع عناصر الأحرف التالية: a,a,a,b,b,b,c,c,c. التنفيذ مشابه جدًا لتابع map المقدَّم سابقًا، وهذه هي الشيفرة البرمجيّة المأخوذ من مكتبة كوتلن القياسيّة:

```
public inline fun <R> CharArray.flatMap(transform: (Char) ->
Iterable<R>): List<R> {
        return flatMapTo(ArrayList<R>(), transform)
    }


    public inline fun <R, C : MutableCollection<in R>>
CharArray.flatMapTo(destination: C, transform: (Char) ->  Iterable<R>): C
{
        for (element in this) {
          val list = transform(element)
          destination.addAll(list)
        }
        return destination
    }
```

تُكرِّر الشيفرة البرمجيّة على كل عناصر المصفوفة المستهدفة ويستدعى التابع `transform()`. وتضاف جميع العناصر التي تُرجع من عملية التحويل إلى المجموعة الوجهة، وربما قد لاحظت بالفعل، فيجب أن تُشتَق الوجهة من `MutableCollection` لأنّنا نحتاج إلى إلحاق العناصر.

توفِّر واجهة برمجة التطبيقات للمكتبة القياسية عددًا من التوابع التي تسمح لك بتحويل مصفوفة إلى نوع تجميعةٍ مختلفٍ، وهذه التوابع هي توابع مُوسّعة تغطي جميع أصناف نوع المصفوفة، وفي ما يلي بعض الأمثلة على كيفيّة تحويل تجميعة من النوع مصفوفة إلى تجميعة أخرى:

```
val longs = longArrayOf(1, 2, 1, 2, 3, 4, 5)
    val hashSet: HashSet<Long> = longs.toHashSet()
    println("Java HashSet:${hashSet.joinToString(",")}")
    val sortedSet: SortedSet<Long> = longs.toSortedSet()
    println("Sorted  Set[${sortedSet.javaClass.canonicalName}]:$
{sortedSet.joinToString (",")}")
    val set: Set<Long> = longs.toSet()
    println("Set[${set.javaClass.canonicalName}]:${set.joinToString(",
")}")
```

```
    val mutableSet = longs.toMutableSet()
    mutableSet.add(10)
    println("MutableSet[${mutableSet.javaClass.canonicalName}]:$
{ mutableSet.joinToString(",")}")
    val list: List<Long> = longs.toList()
    println("List[${list.javaClass.canonicalName}]:${list.joinToString
(",")}")
    val mutableList: MutableList<Long> = longs.toMutableList()
    println("MutableList[${mutableList.javaClass.canonicalName}]:$
{ mutableList.joinToString}")
```

لا أستخدم في العادة نوع المتغيّر في المكان (أي، `val set: Set<Long>`) لكن إذا كنت مبتدئًا في كوتلن، فأوصيك بالقيام بذلك في البداية، فالشيفرة البرمجيّة تعرِّف مصفوفة بسيطة من نوع `Long` وتحوِّلها إلى تجميعات مختلفة (اختلافات قليلة بين `Set` و `List`)، لكل متغيّر، وبصرف النظر عن `HashSet` لجافا، يُكتَب نوع التجميعة الحالية إلى الطرفيّة، وفي ما يلي ما ستطبعه الشيفرة البرمجيّة السابقة:

```
Java HashSet:1,2,3,4,5
    Sorted Set[java.util.TreeSet]:1,2,3,4,5
    Set[java.util.LinkedHashSet]:1,2,3,4,5
    MutableSet[java.util.LinkedHashSet]:1,2,3,4,5,10
    List[java.util.ArrayList]:1,2,1,2,3,4,5
    MutableList[java.util.ArrayList]:1,2,1,2,3,4,5
```

على الرغم من أنك تتعامل مع أنواع كوتلن غير قابلة للتغيير، فإن تجميعة جافا المستخدمة خلف الستار قابلة للتغيير، ولذلك مرّة أخرى، يجري التحقق من عدم قابلية التغيير (immutability) في كوتلن عن طريق تعريف الواجهة.

ما الذي سيحدث برأيك إذا حولت نوع قائمتي إلى النوع `ArrayList` وأضفت عنصرًا إليها؟ مثل:

```
val hackedList = (list as ArrayList<Long>)
    hackedList.add(100)
    println("List[${list.javaClass.canonicalName}]:${list.joinToString
```

```
(",")}")
```

السؤال ليس بخدعة، الشيفرة البرمجية السابقة تُصرِّف وتعمل جيِّدًا، وبعد كل شيء، فنحن في عالم JVM، ولذا، عندما تكون أيدينا على نسخة `ArrayList`، فيمكننا تغيير عناصرها، وسينعكس هذا تلقائيًا من قبل نسخة List لكوتلن، والآن لماذا هذا خطير؟ انظر إلى تابع جافا `next`:

```
public static void dangerous(Collection<Long> l) {
        l.add(1000L);
    }
```

لنفترض أن لديك تابع مكتبة مثل هذا، والذي تستدعيه من داخل شيفرة كوتلن البرمجيّة بالشكل `Arrays.dangerous(list)` وبسبب وضع اسم مستعار للنوع وقت التصريف (تطرقنا لهذا الموضوع في بداية الفصل)، لا يظهر المصرِّف أي خطأ، والمشكلة هي أنك تتعامل مع مجموعة غير قابلة للتغيير في شيفرة كوتلن الخاصة بك، ومع ذلك، بمجرَّد تسليمها إلى شيفرة جافا، سيُخرَق شرط عدم قابلية التغيير، ولذلك إذا كنت تريد الحفاظ على حالة التجميعة، يجب عليك تقديم نسخة من تجميعتك، ولذلك استخدم التابع المُوسّع `toList`.

التعامل مع التجميعات سهل للغاية بفضل واجهة برمجة التطبيقات الغنيّة التي توفرها المكتبة القياسيّة، ومع ذلك، ستحتاج إلى إيلاء المزيد من الاهتمام في البداية قبل أن تكون مألوفة لك. على سبيل المثال، بالنسبة إلى `mutableSet`، سيظهر لك مربع إكمال الشيفرة لبيئة التطير تابعين اثنين هما: `plus` و `plusAssign`، وإذا استدعيت التابع `plus`، فإن مصدر الطقم (set) القابل للتغيير سيبقى في الواقع دون تغيير؛ تكون قيمة الإرجاع هي مجموعة جديدة، وهذه المرة من النوع `Set<Long>`، الغير قابلة للتغيير, وهي غير بديهية قليلًا، ويمكنك أن تجادل أنّه يجب على `plus` المستدعاة مع التجميعات القابلة للتغيير أن تعكس التغيير عليها نفسها، لكن لا تفعل ذلك. ولتطبيق التغيير على مصدر التجميعة نفسها، يجب عليك استخدام `plusAssign`، ونوع الإرجاع هذه المرة هو Unit. والسبب تصرُّف التابع `plus` بهذا الشكل هو أنّه تابع مُوسّع معرّف للنوع `Set`، والذي هو غير قابل للتغيير!

شيء أخير على المصفوفات، قبل الانتقال إلى نوع التجمعية التالي، هل تذكر أن كوتلن تدعم تدمير الكائنات (تحتاج إلى توفير تابع `iterator` جنبًا إلى جنب مع توابع `componentN`)؟ لا يزال هذا صحيح على

المصفوفات، ولقد رأينا في بداية الفصل أن صنف Array يعرِّف التابع iterator، وتـوفِّر التوابـع componentN كتوابع مُوسَّعة:

```
public inline operator fun IntArray.component1(): Int {
        return get(0)
      }
```

ستجد هذه التوابع لكل نوع مصفوفة (IntArray - CharArray - … - Array<T>). قرَّر فريق كوتلن توفير التوابع من component1 إلى component5، ويعـني هـذا أنَّه يمكنـك تفكيـك أول 5 عناصـر فقـط، وهنالـك دائمًا خيار لك لكتابة تابع componentN إضافي، مما يسمح باستعادة عدد أكبر من العناصر عبر التفكيك:

```
val integers = intArrayOf(1, 2, 3, 4, 5, 6)
 val (i1, i2, i3, i4, i5) = integers
 println("i1:$i1; i2:$i2;..;i5=$i5")
```

سـيؤدي تنفيـذ هـذه الشـيفرة البرمجيَّة إلى طباعـة أول 5 عناصـر من مصـفوفة من الأعـداد الصـحيحة، مـاذا سـيحدث، على الـرغم من ذلـك، إذا كنت تفكـك (deconstruct) وحجم المصـفوفة لا يتطـابق مـع العناصـر المستخدمة في التفكيك؟ مثل:

```
val integers = intArrayOf(1, 2, 3)
 val (i1, i2, i3, i4, i5) = integers
```

في هـذه الحالـة، سـتنتهي بـرمي الاسـتثناء java.lang.ArrayIndexOutOfBoundsException، ولذلك، تأكد دائمًا من التحقق من طول المصفوفة قبل تفكيكها.

## 3. القوائم (النوع List)

القوائم (النوع List) هي تجميعـات (collection) مرتَّبـة، فيمكنـك من خلال القائمـة إدراج عنصـر في موقـع محدد، كما يمكنك استرجاع العناصر معلوم الموضع, وتوفّر كوتلن تابعين مدمجين لبناء القوائم القابلة للتغيـير وغـير القابلة للتغيير. تذكر أن عدم قابلية التغيير تتحقَّق عبر واجهة. وإليك طريقة إنشاء قوائم في كوتلن:

```
val intList: List<Int> = listOf
     println("Int  list[${intList.javaClass.canonicalName}]:$
{intList.joinToString(", ")}")

     val emptyList: List<String> = emptyList<String>()
     println("Empty  list[${emptyList.javaClass.canonicalName}]:$
{emptyList.joinToStrin g(",")}")

     val nonNulls: List<String> = listOfNotNull<String>(null, "a", "b",
"c")
     println("Non-Null string lists[${nonNulls.javaClass.canonicalName}]:$
{nonNulls.joinToString (",")}")

     val doubleList: ArrayList<Double> = arrayListOf(84.88, 100.25,
999.99)
     println("Double list:${doubleList.joinToString(",")}")
     val cartoonsList: MutableList<String> = mutableListOf("Tom&Jerry",
"Dexter's Laboratory", "Johnny Bravo", "Cow&Chicken")
     println("Cartoons list[${cartoonsList.javaClass.canonicalName}]:  $
{cartoonsList.joinToString(",")}")

     cartoonsList.addAll(arrayOf("Ed, Edd n Eddy","Courage the Cowardly
Dog"))
     println("Cartoons list[${cartoonsList.javaClass.canonicalName}]: $
{cartoonsList.joinToString(",")}")
```

القوائم الثلاثة الأولى (`intList` و `emptyList` و `nonNulls`) هي نسخ للقراءة فقط، بينما النسختان الأخيرتان قابلتان للتغيير، وبصرف النظر عن قائمة المصفوفات `listOf`، فإن البقية تُرجع نوع كوتلن. بالنسبة لجميع أنواع كوتلن، فإن الشيفرة البرمجيّة تطبع اسم الصنف الحالي المستخدم وقت التشغيل. مخرجات الشيفرة البرمجيّة السابقة هي:

```
Int list[java.util.Arrays.ArrayList]:20,29,40,10
     Empty list[kotlin.collections.EmptyList]:
     Non-Null string lists[java.util.ArrayList]:a,b,c
```

```
Double list:84.88,100.25,999.99
Cartoons list[java.util.ArrayList]: Tom&Jerry,Dexter's Laboratory,Johnny  Bravo,Cow&Chicken
Cartoons list[java.util.ArrayList]: Tom&Jerry,Dexter's Laboratory,Johnny Bravo,Cow&Chicken,Ed, Edd n Eddy,Courage the  Cowardly Dog
```

قد يكون الأمر مفاجئًا، على الرغم من العمل مع أنواع غير قابلة للتغيير، فإن التنفيذ الفعلي يستخدم مجموعة قابلة للتغيير: `ArrayList`، ومن الأشياء المثيرة للاهتمام أيضًا هي أن `listOf` تُرجع النوع `Arrays.ArrayList`. هذا الصنف مختلف عن `java.util.ArrayList`، فالأول على الرغم من أنه مشتق من الصنف `Collection`، فلا يمكن تغييره عن طريقة إضافة عناصر أو حذفها، فسينتهي كلاهما برمي `UnsupportedOperationException`. ومع ذلك، فهي ليست مجموعة غير قابلة للتغير بشكل كامل لأنه يمكنك استبدال عنصر ذي موقع معروف داخل تجميعتك، فلذلك، يحقق نظام أنواع كوتلن ثباته من خلال تعريف الواجهة، لكن لا شيء يمنعك من القيام بما يلي:

```
(intList as AbstractList<Int>).set(0, 999999)
    println("Int  list[${intList.javaClass.canonicalName}]:${intList.joinToString(", ")}")

    (nonNulls as java.util.ArrayList).addAll(arrayOf("x", "y"))
    println("countries  list[${nonNulls.javaClass.canonicalName}]:${nonNulls.joinToString( ",")}")

    val hacked: List<Int>= listOfNotNull(0,1)
    CollectionsJ.dangerousCall(hacked)
    println("Hacked list[${hacked.javaClass.canonicalName}]:${hacked.joinToString(",") }")

    //Java code
    public class CollectionsJ {

      public static void dangerousCall(Collection<Integer> l) {
```

```
            l.add(1000);
        }
    }
```

في المثال الأول، تتحوّل التجميعة إلى الصنف الأب `Arrays.ArrayList: AbstractList` ولا يمكنك تحويلها إلى `Arrays.ArrayList` لأن الصنف موسوم بأنَّه خاص (private) في JDK. بمجرَّد تغيير النوع، يمكننا استخدام التابع set وتغيير أول عدد صحيح ببساطة بواحد جديد: 999999.

في المثال الثاني، حولنا متغيِّر جافا `ArrayList` واستخدمنا التوابع المكشوفة لتعديل التجميعة. المثال الأخير هو مشكلة غير متوقعة نموذجيّة، فعند العمل داخل سياق JVM، فأنت ملزم باستخدام مكتبات خارجية، فإذا قمت بتسليم مرجع تجميعة غير قابلة للتغيير لكوتلن، فلا يمكن ضمان تحقق شرط منع التغيير بعد الآن. إذا لم تحتاج إلى تغيير تجميعتك، فيجب عليك تمرير لقطة (snapshot)، ولذلك، استخدم `hacked.toList` لحل المشكلة.

لقد رأيت كيفية إنشاء القوائم (تذكر أنه يمكنك أيضًا تحويل تجميعات من أنواع أخرى إلى قوائم عن طريق التابع `ToList.`)، لكن دعنا الآن نلقي نظرة على بعض الأمثلة البسيطة لنعرض بعض التوابع المُوسّعة المتوفرة في المكتبة:

```
data class Planet(val name: String, val distance: Long)

     val planets = listOf( Planet("Mercury", 57910000), Planet("Venus",
108200000), Planet("Earth", 149600000), Planet("Mars", 227940000),
Planet("Jupiter", 778330000), Planet("Saturn", 1424600000),
Planet("Uranus", 2873550000), Planet("Neptune", 4501000000),
Planet("Pluto", 5945900000))

     println(planets.last())          //Pluto
     println(planets.first())         //Mercury
     println(planets.get(4))          //Jupiter
     println(planets.isEmpty())       //false
```

```
    println(planets.isNotEmpty())    //true

    println(planets.asReversed())        //"Pluto", "Neptune"
    println(planets.elementAtOrNull(10)) //Null
```

يُعرِّف هذا المقتطف من الشيفرة قائمة من كواكب مجموعتنا الشمسيّة وبُعدها عن الشمس، وباستخدام هذه القائمة كهدف، يمكنك رؤية عمل التوابع الأساسيّة، لن أشرح كل واحد منها لأن اسم التابع يصف عمله.

دعنا ننتقل إلى عمليات أكثر تعقيدًا على القوائم، لنفترض أنك ترغب في دمج تجميعة مع تجميعة أخرى، فتوفر المكتبة دعمًا لمثل هذه العملية عن طريق التابع `zip`.، فنضيف في المثال التالي قائمة الكواكب `planets` إلى مصفوفة تحتوي على قطر كل كوكب:

```
planets.zip(arrayOf(4800, 12100, 12750, 6800, 142800, 120660,  51800,
49500, 3300))
      .forEach {
         val (planet, diameter) = it
         println("${planet.name}'s diameter is $diameter km")
      }
```

نفِّذ الشيفرة وستطبع قطر كل كوكب، أراهن على أنك تتساءل، ماذا يحدث إذا كان حجم المجموعتين مختلف؟ لنفترض أننا حذفنا قطر بلوتو أي `Pluto`، ففي هذه الحالة، ستزيل عملية `join` كوكب بلوتو `Pluto`.

لقد استلهمت مكتبة `Collection` من مكتبة `collection` المقابلة في لغة سكالا وتأثرت بها تأثرًا كبيرًا، فهي تأتي بدعم التابعين `foldLeft` و `foldRight`، وينبغي أن تكون هذان التابعان مألوفين لأي مطوّر سكالا، فهما مراكمات (accumulators)، أي تأخذ قيمة أوليَّة وتكرِّر تنفيذ دالة لامدا (من اليسار إلى اليمين أو من اليمين إلى اليسار) على كل عنصر من عناصر التجميعة المستهدفة مع مراكمة النواتج، وتُرجع القيمة التراكمية النهائية. لنفترض أنَّنا نريد سرد الكواكب من الأبعد إلى الأقرب إلى الشمس، هنالك طريقة واحدة لتحقيق هذا عبر `foldRight`:

```
val reversePlanetName = planets.foldRight(StringBuilder()) {
```

```
        planet, builder -> builder.append(planet.name)
        builder.append(";")
}
println(reversePlanetName)   //Pluto, Neptune..Earth;Venus;Mercury
```

لعرض `foldLeft`، دعنا ننتقل إلى مشكلة مختلفة النطاق؛ لنفترض أنك تملك بطاقة إلكترونية وترغب في حساب سعر جميع العناصر في الموضوعة عربة التسوق، ولهذا يوفر `foldLeft` لك وسيلة لحساب السعر الإجمالي:

```
data class ShoppingItem(val id: String, val name: String, val  price:
BigDecimal, val quantity: Int)

     val amount = listOf( ShoppingItem("1", "Intel i7-950 Quad-Core
Processor", BigDecimal("319.76"), 1), ShoppingItem("2", "Samsung  750 EVO
250 GB 2.5 inch SDD", BigDecimal("71.21"), 1))
     .foldRight(BigDecimal.ZERO) {
        item, total -> total + BigDecimal(item.quantity) * item.price
     }
     println(amount)   //390.97
```

تحصل جميع أنواع القوائم على دعمٍ للتابعين الموسّعين `map` و `flatMap`، وهما الأكثر استخدامًا في واجهة برمجة التطبيقات للمكتبة القياسية عندما يتعلّق الأمر بالتلاعب في المجموعة:

```
planets.map { it.distance }   //List(57910000, ...,5945900000)

     val list = listOf(listOf(10, 20), listOf(14, 18), emptyList())
     val increment = { x: Int -> x + 1 }
     list.flatMap { it.map(increment) }   //11,21,15,10
```

يستخرج السطر الأول في المثال تجميعة أخرى (`List<Long>` لنكون دقيقين) من التجميعة `planets`، فتحتوي هذه المجموعة الجديد على المسافة من الشمس لكل كوكب في نظامنا الشمسي، سهل جدًا!

الجزء الثاني من المثال أكثر تطورًا، فيبدأ بقائمة من الأعداد الصحيحة ثم يعرِّف دالة لامدا لزيادة معامل العدد الصحيح بواحد، السطر الأخير من الشيفرة البرمجيّة تطبّق لامدا لكل عنصر من كل قائمة من القوائم الثلاثة ومن ثم

يسطّح `flattens` المجموعة الناتجة، وتصبح قائمة قوائم الأعداد الصحيحة قائمة واحدة من الأعداد الصحيحة أي دون أي تشعُّب.

يطبّق كائن التفكيك على القوائم كذلك، فكما هو الحال مع المصفوفات، ستحصل على دعم مبتكر لتفكيك العناصر الخمسة الأولى من القائمة، وهذه الشيفرة البرمجيّة مشابهة لتلك المستخدمة للمصفوفات:

```
val chars = listOf('a', 'd', 'c', 'd', 'a')
val (c1,c2,c3,c4,c5) = chars
println("$c1$c2$c3$c4$c5")//adcda
```

سوف أختتم قسم القوائم بإظهار كيف يمكنك تحويل قائمة إلى نوع تجميعة مختلف:

```
val array: Array<Char> = chars.toTypedArray()
val arrayBetter: CharArray = chars.toCharArray()
val set: Set<Char> = chars.toSet()              //[a,d,c]
val charsMutable: MutableList<Char> = chars.toMutableList()
```

يوفر المثال خيارين لتحويل قائمة إلى مصفوفة، في قسم المصفوفات، تعلمت الفرق بين `Array<T>` و `IntArray` و `DoubleArray` وغيرها، ولماذا من الأفضل استخدام تنفيذ الأنواع الأساسية، ويُطبَّق نفس المنطق على هذا التحويل كذلك، ولذلك، من الأفضل استخدام `to***Array` عند التعامل مع الأنواع الأساسية.

## 4. الخرائط (النوع Map)

تسمح لك تجميعة الخرائط[13] (map collection)، بربط كائن (يدعى مفتاح) بكائن آخر (القيمة المرتبطة بذلك المفتاح)، ويشترط هذا النوع أن يكون المفتاح فريدًا ومرتبطًا بقيمة واحدة على الأكثر. الجزء المثير للاهتمام حول الخريطة هو أن الواجهة توفِّر ثلاثة طرائق لعرض محتوى هذا النوع من التجميعات: طقم (set) من المفاتيح، تجميعة من القيم، وطقم (set) من الأزواج مفاتيح-قيم.

13 هذا هو الاسم المترجم الشائع لهذا النوع، النوع Map، ولكنني -يقول المحرِّر- أفضل أن أدعوه باسم «الرابطة» لأنه نوع قائم على الربط بين قيمتين مع بعضهما بعضًا، أي ربط قيمة بمفتاحها.

عند استخدام الخريطة، ستحتاج إلى الانتباه إلى المفاتيح التي تستخدمها، فعند إضافة عنصر (العنصر هنا هو زوج من مفتاح-قيمة) إلى خريطة، فأول شيء تفعله هو تحديد الموضع الذي يجب أن يضاف إليه، للقيام بذلك، يُستخدَم التابع `hashCode`، وبعد ذلك، وبالاعتماد على التنفيذ، يُستخدَم التابع `equals`. لذلك، يجب أن تكون المفاتيح غير قابلة للتغيير، وإلا فلا يمكن تحديد سلوك الخريطة.

نحن نعرف بالفعل أن كوتلن توفِّر دعمًا للخرائط القابلة للتغيير وغير القابلة للتغيير على مستوى الواجهة، وينعكس هذا على واجهة برمجة تطبيقات التجميعات نظرًا لوجود توابع محددة لكل منها:

```
data class Customer(val firstName: String, val lastName: String,  val id:
Int)

     val carsMap: Map<String, String> = mapOf("a" to "aston martin",  "b"
to "bmw", "m" to "mercedes", "f" to "ferrari")
     println("cars[${carsMap.javaClass.canonicalName}:$carsMap]")
     println("car maker starting with 'f':${carsMap.get("f")}")
//Ferrari
     println("car maker starting with 'X':${carsMap.get("X")}")  //null

     val states: MutableMap<String, String>= mutableMapOf("AL" to
"Alabama", "AK" to "Alaska", "AZ" to "Arizona")
     states += ("CA" to "California")
     println("States [${states.javaClass.canonicalName}:$states")
     println("States keys:${states.keys}")//AL, AK, AZ,CA
     println("States values:${states.values}")//Alabama, Alaska,  Arizona,
California

     val customers: java.util.HashMap<Int, Customer> = hashMapOf(1 to
Customer("Dina", "Kreps", 1),  2 to Customer("Andy", "Smith", 2))

     val linkedHashMap: java.util.LinkedHashMap<String, String> =
linkedMapOf("red" to "#FF0000","azure" to "#F0FFFF","white" to  "#FFFFFF")

     val sortedMap: java.util.SortedMap<Int, String> = sortedMapOf(4 to
```

```
"d", 1 to "a", 3 to "c", 2 to "b")
    println("Sorted  map[${sortedMap.javaClass.canonicalName}]:$
{sortedMap}")
```

يعيد لك أول بانيين نوع كوتلن، في حين أن البانيات الثلاثة الأخيرة تعيد تنفيذ `map` `util` لجافا. إذا نفَّذت الشيفرة البرمجيَّة، فستحصل على مخرجات أنواع الخريطة `Map` لكوتلن وصنف `Class` جافا المستخدمة كتنفيذ (implementation)، وفي كلتا الحالتين، فذاك الصنف هو `LinkedHashMap`. أنا متأكد من أنك غالبًا تعرف الفرق بين أنواع الخريطة الثلاثة، فدعنا نتذكرها سوية:

- `HashMap`: هو تنفيذ يعتمد على الجدول لواجهة `map`، ففي حين يسمح باستعمال القيم الفارغة `null` مع المفتاح أو القيمة، فلا يضمن الصنف ترتيب العناصر أو حقيقة أنَّها ستبقى ثابتة مع مرور الوقت، فهذا التنفيذ لديه وقتًا ثابتًا (مستهلكًا للأداء) لتنفيذ التابعين `get` و `put`، فلو افترضنا أن دالة `hash` توزِّع العناصر بشكل صحيح بين الدلاء (buckets)، فيحتفظ الصنف عامل الحمولة كمقياس لمدى امتلاء الخريطة قبل زيادة حجمها (سعتها)، فعندما يتجاوز عدد المدخلات في جدول `hash` حاصل ضرب عامل الحمولة والسعة الحاليَّة، فسيعاد تخطيط جدول map (أي يعاد بناء هيكل البيانات الداخلية) بحيث يحتوي جدول `hash` على ضعف عدد الدلاء.
- `LinkedHashMap`: هو مزيج من `HashMap` وتنفيذ قائمة مترابطة (linked list) لواجهة `map`، مع ترتيب يمكن التنبؤ به. يختلف هذا التنفيذ عن `HashMap` في أنه يحتفظ بقائمة متصلة مضاعفة تعمل من خلال جميع الإدخالات، وتعرِّف هذه القائمة المترابطة ترتيب العناصر، والذي يكون عادةً الترتيب الذي تدخل فيها المفاتيح إلى الخريطة، إذ لا يتغيّر ترتيب الإدخال عندما يعاد إدخال مفتاح إلى الخريطة.
- `TreeMap`: هو تنفيذ لخريطة `map` يعتمد على تنفيذ شجرة الأحمر-الأسود (red-black tree)، تُرتّب الخريطة `map` استنادًا إلى الترتيب الافتراضي لمفاتيحها أو من خلال الموازنة المقدمة في وقت إنشاء الخريطة اعتمادًا على الباني المُستخدم. يوفِّر هذا التنفيذ ضمان كون وقت التنفيذ هو `log(n)` لعمليات التحقق من وجود مفتاح `containsKey` والجلب `get` والإضافة `put` والحذف `remove`. تعد شجرة الأحمر-الأسود حالةً خاصةً من شجرة البحث الثنائيّة (binary search tree) والتي لكل عقدة فيها لونًا واحدًا (أحمر أو أسود) مرتبط بها (بالإضافة إلى مفتاحها وأبنائها المتوضعين اليمين وعلى اليسار).

تخضع بنية الشجرة بالقواعد التالية: عقدة الجذر سوداء، أبناء العقدة الحمراء هي سود، عدد العقد السوداء في المسار من الجذر إلى الابن الفارغ (null) هي نفسها.

بما أن هذا الكتاب لا يركز على هياكل البيانات، أعتقد أن هذه المعلومات كافية عن تنفيذات الخريطة، يمكنك دائما القيام بمزيد من الأبحاث لتوسيع مداركك بهذه التنفيذات وإجابيات وسلبيات استخدام كل تنفيذ منها.

ذكرنا بالفعل، بالنسبة إلى القوائم، أنه بمجرّد تمرير مرجعك إلى مكتبة جافا، فإن عدم قابلية التغيير ستكون خارج السيطرة، وينطبق نفس الشيء على أي أنواع map في كوتلن، ففي الشيفرة البرمجيّة التالية، يمكنك مشاهدة مثال بسيط لدالة جافا تأخذ خريطة من السلاسل النصية، وكل ما تفعله هو إضافة عنصر جديد (يمكن بسهولة إزالة واحد منها أو مسح الخريطة بأكملها). عند استدعاء الشيفرة البرمجيّة من كوتلن، سترى أن بيئة التطوير المتكاملة IDE تظهر لك نوع المنصّة `(Mutable)Map<String!,String!>` لذلك تحقق دائمًا مما يفعله استدعاء الشيفرة البرمجيّة:

```
public static void dangerousCallMap(Map<String,String> map){
        map.put("newKey!", "newValue!");
}
CollectionsJ.dangerousCallMap(carsMap)
println("Cars:$carsMap")   //Cars:a=aston martin, b=bmw, m=mercedes,
f=ferrari, newKey!=newValue!
```

إذا كنت ترغب في تجنّب تغيير مجموعة map، فستحتاج إلى أخذ لقطة (snapshot) من الخريطة وتسليمها إلى تابع جافا، في حين أنّها ليست أجمل شيفرة برمجيّة، إلا أنها تفي بالغرض: `carsMap.toList().toMap()`.

لنلق نظرة الآن على بعض التوابع المُوسّعة المتوفرة للنوع Map:

```
customers.mapKeys { it.toString() }   // "1" =
Customer("Dina","Kreps",1),
customers.map { it.key * 10 to it.value.id }         // 10= 1, 20  =2
customers.mapValues { it.value.lastName }  // 1=Kreps, 2="Smith
customers.flatMap { (it.value.firstName +
it.value.lastName).toSet() }.toSet() //D, i, n, a, K, r, e, p, s,  A, d,
y, S, m, t, h]
```

```
linkedHashMap.filterKeys { it.contains("r") }  //red=#FF0000,
states.filterNot { it.value.startsWith("C") } //AL=Alabama,  AK=Alaska,
AZ=Arizona
```

يسمح لك المثال الأول بتغيير نوع أحد المفاتيح، بينما تشير `it` إلى كامل نسخة `Map.Entry`، لن يغيّر هذا التابع نوع القيم، إذا أرجعت دالة لامدا نفس القيمة أكثر من مرّة، فستفقد العناصر، وستبقى القيمة الأخيرة فقط. تخيّل لو قمنا بإرجاع قيمة ثابتة من دالة، فإن الخريطة الناتجة ستحتوي على عنصر واحد. يسمح المثال الثاني المستدعي بتغيير كل من المفاتيح ونوع القيم، المثال الثالث على عكس المثال الأول، يمكنك فيه إرجاع نفس القيمة دون التأثير على حجم المجموعة. سينتهي بك المطاف مع مجموعة قيم تظهر فيها بعض العناصر أكثر من مرّة. تذكّر، ستُرجع الدالة `flatMap` في المكتبة القياسيّة `List<T>`، في الشيفرة البرمجية للمثال السابق، ستُحدّد جميع الأحرف المستخدمة في `customers` بالأسماء، سيظهر آخر تابعين لك كيف يمكنك اختيار عناصر الخريطة بناءً على مُرشّح (filter)، في كلتا الحالتين، سينتهي بك الأمر مع نسخة خريطة يحتوي على العناصر التي تفي بالمعايير الخاصة بك.

## 5. الأطقم: (النوع Set)

تعد الأطقم (النوع Set)ـ أحد أنواع التجميعات، إذ هي تجميعة من عناصر فريدة، وهذا يعني أنه لا يمكنك وضع عنصرين متساويين بالقيمة أحدهما باسم `i1` والآخر باسم `i2` (أي إذا كان `i1==i2` والذي يُترجَم إلى `i1.equals(i2) == true`) ويُطبَّق نفس المنطق على مرجع العدم (null reference)، فلا يمكنك تخزين أكثر من عنصر `null` في طقم.

لإنشاء نسخ من التجميعة Set، يمكنك استخدام أي من الطرائق الموضحة في المثال التالي:

```
data class Book(val author: String, val title: String, val year:  Int, val
isbn: String)


    val intSet: Set<Int> = setOf(1, 21, 21, 2, 6, 3, 2)  //1,21,2,6,3
    println("Set of  integers[${intSet.javaClass.canonicalName}]:
$intSet")
```

```
    val hashSet: java.util.HashSet<Book> = hashSetOf(
        Book("Jules Verne", "Around the World in 80 Days Paperback",
2014, "978-1503215153"),
        Book("George R.R. Martin", "Series: Game of Thrones: The Graphic
Novel (Book 1)", 2012, "978-0440423218"),
        Book("J.K. Rowling", "Harry Potter And The Goblet Of Fire (Book
4) Hardcover", 2000, "978-0439139595"),
        Book("Jules Verne", "Around the World in 80 Days Paperback",
2014, "978-1503215153")
)  //Jules Verne,  J.K. Rowling,George R.R. Martin
    println("Set of books:${hashSet}")

    val sortedIntegers: java.util.TreeSet<Int> = sortedSetOf(11, 0, 9,
11, 9, 8)  //0,8,9,11
    println("Sorted set of integer:${sortedIntegers}")

    val charSet: java.util.LinkedHashSet<Char> = linkedSetOf('a', 'x',
'a', 'z', 'a')   //a,x,z
    println("Set of characters:$charSet")

    val longSet: MutableSet<Long> = mutableSetOf( 20161028141216,
20161029121211, 20161029121211) //20161028141216, 20161029121211
    println("Set of longs[${longMutableSet.javaClass.canonicalName}]  :
$longSet")
```

يمكنـــك رؤيـــة نتيجـــة كـــل طقم في التعليقـــات؛ تُرجِـــع التابعـــان المُوسّـــعان setOf و mutableSetOf فقط نوع كوتلن، وتُستخدَم التوابع الثلاثة المتبقية في توليد نوع جافا. إذا نفَّذت شيفرتك البرمجيّة، فسترى أن التنفيذات المكتوبة للنوع Set القابلة والغير في كوتلن مجسمة عبر LinkedHashSet، والذي هو قابل للتغيير بالتأكيد. ولفهم الفروقات بين مختلف التنفيذات المُوفَّرة، دعنا نرى ما يقوله JDK على كل واحدة منها:

- LinkedHashSet: تنفيذات جدول hash والقائمة المترابطة (linked list) لواجهة التجميعة Set مع ترتيب يمكن التنبؤ بـه. يختلف هذا التنفيذ عن التنفيذ HashSet (التالي) في أنَّه يحتفظ بقائمة

مترابطة مزدوجة (doubly-linked list) تعمل من خلال جميع مدخلاتها. تُعرِّف القائمة المترابطة الترتيب المُكرَّر (iteration ordering)، وهو الترتيب الذي أُدرجَت فيه العناصر في الطقم. يجنّب هذا التنفيذ الترتيب الفوضي الذي يقدّمة التنفيذ HashSet، مبتعدًا عن أي تكلفة للأداء التي ترتبط بالتنفيذ TreeSet.

- HashSet: تُنفَّذ واجهة الطقم Set بدعم من جدول hash (نسخة HashMap في الواقع)، ولا يقدم أي ضمانات بشأن ترتيب عناصر الطقم، فلا يضمن أن الترتيب سيظل ثابتًا بمرور الوقت، ويقدّم هذا التنفيذ وقت أداء ثابت للعمليات الأساسيّة (الإضافة add والحذف remove والتحقق من الاحتواء contains والحجم size) بافتراض أن دالة hash تشتت العناصر بشكل صحيح بين الدلاء (أي map buckets).
- TreeSet: هو تنفيذ للطقم Set مبني على أساس التنفيذ TreeMap الذي شرحناه آنفًا، فترتِّب العناصر باستخدام ترتيبها الطبيعي، أو من خلال الموازنة المقدّمة في وقت إنشاء الطقم، أي اعتمادًا على الباني المُستخدَم، ويُوفِّر التنفيذ الوقت log(n) لإجراء العمليات الأساسيَّة (الإضافة add والحذف remove والتحقق من الاحتواء contains).

إن تغطية كل تابع متوفّر على واجهة set يتجاوز نطاق هذا الفصل، ومع ذلك، سنعرض بعض التوابع المتاحة، يمكنك دائمًا التقاط الوثائق والتعرّف على مجموعة كاملة من التوابع المكشوفة:

```
println(intSet.contains(9999))   //false
     println(intSet.contains(1))     //true
     println(books.contains(Book("Jules Verne", "Around the World in 80 
Days Paperback", 2014, "978-1503215153"))) //true
     println(intSet.first())          //1
     println(sortedIntegers.last())   // 11
     println(charSet.drop(2))         // z
     println(intSet.plus(10))         // 1,21,2,6,3,10
     println(intSet.minus(21))         // 1,2,6,3
     println(intSet.minus(-1))         // 1,21,2,6,3
     println(intSet.average())         // 6.6
```

```
    println(longSet.plus(11))     // 20161028141216, 20161029121211
    println(longSet)         //20161028141216, 20161029121211
```

يمكنك رؤية المخرجات في التعليقات، يجب أن تكون أسماء التوابع (الأجنبية) كافية لإعطائك معلومات عما تقوم به. لاحظ أنَّ التابعين `plus` و `minus` لا يغيِّران التجميعة، إذ هذان التابعان المُوسّعان مُعرَّفان الواجهة `Set` الغير قابلة للتغيير، وبالتالي، سينتهي بها الأمر إلى إنشاء تجميعة طقم ثابتة جديدة.

لا يمكننا التحدث عن نوع التجميعة دون التحدث عن التابعين الموسّعين: `map` و `flatMap`؛ ففي المثال الأول، تستخرج الشيفرة البرمجيّة زوج author-title من طقم الكتب (set of books) بينما يحصل المثال الثاني على جميع الأحرف المستخدمة في عناوين الكتب في الخريطة (map):

```
println(books.map{Pair(it.author,it.title)})    // Jules Verne- Around the World in 80 Days Paperback,
    println(books
       .flatMap { it.title.asIterable() }
       .toSortedSet()
    )    //[ , (, ), 0, 1, 4, 8, :, A, B, D, F, G, H, N, O, P, S, T,  W, a, b, c, d, e, f, h, i, k, l, m, n, o, p, r, s, t, u, v, y]
```

كما رأينا مع التجميعات الأخرى، توفِّر المكتبة القياسية توابع مُوسّعة لتحويل تجميعة طقم إلى نوع تجميعة أخرى، هنالك عدد غير قليل من التوابع المُوسّعة لإنجاز ذلك، كما هو موضَّح في المثال التالي:

```
val longsList: List<Long> =longSet.toList()
val longsMutableList = longSet.toMutableList()
val donot= longSet.toLongArray()
val rightArray = longSet.toTypedArray()
```

كُتِب هذا المثال عن قصد لإعادة تكرار عملية التحويل إلى مصفوفات؛ في حين أن هنالك خياران للأنواع الأساسية، تأكد من اختيار `to***Array` للحصول على أفضل أداء.

## 6. العرض في وضع القراءة فقط

سيصادفك في كوتلن مفهوم العرض في وضع القراء فقط (Read-only view) لتجميعة قابلة للتغيير، وربما

ستتساءل ما هو الفرق بينها وبين التجميعة الغير قابلة للتغيير؟ من الأسهل الإجابة عن سؤالك هذا بمثال عملي، ولذلك، لننشئ قائمة قابلة للتغيير من السلاسل النصية:

```
val carManufacturers: MutableList<String> =  mutableListOf("Masserati",
"Aston  Martin","McLaren","Ferrari","Koenigsegg")
     val carsView: List<String> = carManufacturers

     carManufacturers.add("Lamborghini")
     println("Cars View:$carsView")  //Cars View: Masserati, Aston Martin,
McLaren,  Ferrari, Koenigsegg, Lamborghini
```

تهيئ الشيفرة البرمجيّة قائمة قابلة للتغيير لمصنعي السيارات ثم تقدّم عرضًا (view) لها عبر المتغير `carsView`، إذ أبقينا مرجعًا إلى المتغيّر الأخير فقط، فيمكننا في الواقع عد التجميعة غير قابلةٍ للتغيير تمامًا (immutable)، ومن هنا يأتي مصطلح العرض في وضع القراءة فقط. ومع ذلك، إذا لم يكن الأمر كذلك، فأي تغييرات تُجرَى على التجميعات الأساسية ستنعكس على العرض تلقائيًا. يتحقَّق العرض عن طريق تحويل التجميعة إلى الواجهة `List` غير القابلة للتغيير، وتذكر أنَّ التنفيذات وقت التشغيل قابلة للتغيير.

## 7. الوصول المفهرس

تسهِّل كوتلن الوصول إلى عناصر قائمة أو إرجاع القيم لمفتاح في خريطة، ولذلك لا حاجة لك لتوظيف أسلوب جافا باستعمال التابع `get(index)` أو التابع `get(key)` على التوالي، لكن يمكنك ببساطة استخدام أسلوب الأقواس `[]` الذي تشتهر به المصفوفات للوصل إلى العناصر:

```
val capitals = listOf("London", "Tokyo", "Instambul", "Bucharest")
     capitals[2]   //Tokyo
     //capitals[100] java.lang.ArrayIndexOutOfBoundException

     val countries = mapOf("BRA" to "Brazil", "ARG" to "Argentina",  "ITA"
to "Italy")
     countries["BRA"]   //Brazil
     countries["UK"]    //null
```

في حين أنَّه يوفِّر عليك بعض الكتابة، أجد أنَّ هذا البناء أسهل وأكثر وضوحًا من ناحيـة قـراءة الشـيفرة، لكن لا شيء يمنعك من العودة مرَّة أخرى إلى التابع `get`.

تتـوفَّر الصـياغة السـابقة في كـوتلن فقـط، ويكمن السـبب في تصـريح الواجهـة للنـوع `List` والنـوع `Map`، تم سردها في بداية هذا الفصل، ولذلك يمكنك العثور على التعريف التالي:

```
//list
    public operator fun get(index: Int): E

//map
    public operator fun get(key: K): V?
```

بمـا أنَّه صُـرِّح عن التـابعين على أنهمـا عـاملين أي `operator`، فيمكننـا اسـتخدام المصـفوفة مثـل الفهرسـة (indexing) لاختصار كتابة `get`..

## 8. المتتالية (النوع Sequence)

عرّفنـا مـا هي المتتاليـة (النـوع Sequence) ومـاذا تفعـل في بدايـة هـذا الفصـل، ويسـتعمل هـذا النـوع من التجميعـات عنـدما لا يُعـرَف حجم التجميعـة مقـدمًا. فكـر مثلًا في قـراءة جـدول من قاعـدة البيانـات لا تعـرف عـدد السجلات فيه وبالتالي لا تعرف العدد الذي ستحصل عليـه، أو عنـد قـراءة ملـف `csv.` محلي لا تعـرف عـدد الأسـطر الـتي يحتويهـا. يمكنـك التفكـير في المتتاليـة على أنَّهـا قائمـة تكـبر مـع الـوقت؛ تقيَّم المتتاليـة على أسـاس قاعـدة «المعرفـة عنـد الحاجـة» (need-to-know)، وفقـط في النقطـة المطلوبـة؛ فكِّر مثلًا في سلسـلة فيبوناتشـي، فلا حاجـة إلى إنشـاء تجميعـة مسـبقًا، فكم عـدد العناصـر الـتي تحتـاج إلى حسـابها؟ لا أحـد يعـرف، ويحـدَّد ذلـك عنـد الاستدعاء.

إذا استخدمت سكالا أو جافا 8، فسترى أن النوع `Sequence` هو نوع كوتلن المكافئ للنوع `Stream`، وبمـا أن كوتلن تدعم جافا 6 الذي لا يدعم مكتبات المجاري التدفقية (Streaming)، كـان لزامًـا عليهـا العمـل على نسـختها الخاصة، وللأسف، فلا تأتي مكتبة كوتلن مع دعم لمعالجة المتتالية المتوازية (parallel sequence).

قبل المضي قدمًا في بعض الأمثلة، إليك عدة طرائق لإنشاء متتالية:

```
val charSequence: Sequence<Char> =  charArrayOf('a','b','c').asSequence()
//a,b,c
     println("Char  sequence:[${charSequence.javaClass.canonicalName}]:$
{charSequence. joinToString(",")}")
println("Char  sequence:[${charSequence.javaClass.canonicalName}]:$
{charSequence. joinToString(",")}")
     println("Char  sequence:[${charSequence.javaClass.name}]:$
{charSequence.joinToStr ing(",")}")


     val longsSequence: Sequence<Long> = listOf(12000L, 11L, -
1999L).asSequence()  // 1200,11,-1999
     println("Long  sequence:[${longsSequence.javaClass.canonicalName}]:$
{longsSequenc e.joinToString(",")}")
     println("Long  sequence:[${longsSequence.javaClass.name}]:$
{longsSequence.joinToS tring(",")}")


     val mapSequence: Sequence<Map.Entry<Int, String>> =  mapOf(1 to  "A",
2 to "B", 3 to "C").asSequence() //1=A,2=B,3=C
     println("Long  sequence:[${mapSequence.javaClass.canonicalName}]:$
{mapSequence.jo inToString(",")}")
     println("Long  sequence:[${mapSequence.javaClass.name}]:$
{mapSequence.joinToStrin g(",")}")


     val setSequence: Sequence<String> = setOf("Anna","Andrew", "Jack",
"Laura","Anna").asSequence()
     println("String  sequence:[${setSequence.javaClass.canonicalName}]:$
{setSequence.jo inToString(",")}")  //Anna, Andrew,Jack, Laura
val intSeq = sequenceOf(1, 2, 3, 4, 5)
     println("Sequence of  integers[${intSeq.javaClass.canonicalName}]:
$intSeq")


     val emptySeq: Sequence<Int> = emptySequence<Int>()
     println("Empty  sequence[${emptySeq.javaClass.canonicalName}]:
$emptySeq")
```

```
    var nextItem = 0
    val sequence = generateSequence {
      nextItem += 1
      nextItem
    }
    //    sequence.joinToString(",") -> don't! Out of memory will be
thrown
    println("Unbound int  sequence[${sequence.javaClass.canonicalName}]:$
{sequence.takeWhile  {
      it <100
     }.joinToString(",")}") //1,2,3...99
    // println("Unbound int  sequence[$
{sequence.javaClass.canonicalName}]:${sequence.takeWhile  {
      it < 100
    }.joinToString(",")}")  //java.lang.IllegalStateException: This
sequence can be consumed  only once.

    val secondSequence = generateSequence(100) { if ((it + 1) % 2 ==  0)
it + 1 else it + 2 }
val secondSequence = generateSequence(100) { if ((it + 1) % 2 ==  0) it +
1 else it + 2 }

    println("Unbound int  sequence[$
{secondSequence.javaClass.canonicalName}]:${secondSequen ce.takeWhile {
      it <110
    }.toList()}") //100, 102, 104, 106, 108]
```

يمكن تحويل جميع أنواع التجميعات التي رأيناها حتى الآن إلى متتالية؛ الجزء المثير للاهتمام هو المخرجات لاسم الصنف. نفِّذ الشيفرة السابقة وسترى أنَّ المخرجات في الطرفيَّة هي العَدم null عندما يتعلق الأمر بالاسم الأساسي للصنف، ومع ذلك، عندما نطبع الاسم سنرى شيئًا مثل:

```
kotlin.collections.ArraysKt___ArraysKt$asSequence$inlined$Sequence$9
```

هـل تتسـاءل لمـاذا؟ عنـدما نطلـع على الشـيفرة سـنفهم كـل شـيء. هـذا مـا يحصـل فعلًا عنـد اسـتدعاء asSequence:

```
public fun <T> Iterable<T>.asSequence(): Sequence<T> {
        return Sequence { this.iterator() } }

     public inline fun <T> Sequence(crossinline iterator: () ->
Iterator<T>): Sequence<T> = object : Sequence<T> {
        override fun iterator(): Iterator<T> = iterator()
     }
```

لدينا واجهة Sequence، ولدينا أيضًا التـابع المُوسّـع Sequence مع معامـل واحـد وهـو دالـة تُرجـع مُكـرِّرًا (iterator)، فجميع الأنواع مشتقة من الواجهة Iterable، وفي المقابل، توفِّر تابعًا مثل iterator() وبالتالي فـإنَّ asSequence تسـتفيد منـه. ينشـئ التـابع Sequence نسـخةً لصـنف مجهـول يـرث الواجهـة Sequence، ويسـتخدم الوسـيط iterator لإرجـاع المُكـرِّر. توجـد توابـع مُوسّـعة خاصـة بالمصـفوفات بمـا أنَّهـا لا تُشـتَق من Iterable؛ ولمّا كنا نستخدم CharArray، فتُغطي الشيفرة البرمجية التالية للمكتبـة القياسـية هـذا النـوع فقط، لكن توجد تنفيذات مكافئة لأنواع المصفوفات الأخرى:

```
public fun CharArray.asSequence(): Sequence<Char> {
        if (isEmpty()) return emptySequence()
        return Sequence {
          this.iterator()
         }
     }
```

لإنشاء متتالية ذات طول ثابت، يمكنك دائمًا الاعتماد على التابع المُوسّـع sequenceOf. ولمّا كـان هـذا التـابع يأخذ قائمة متغيِّرة الطول من المعاملات، فيسـتخدم شـيفرة Array<T>.asSequence، وبالنسـبة لسـيناريوهات يكون فيها قيد الحد الأعلى للمتتالية غير معروف، فيمكنك استخدام التـابع generateSequence. يُرجـع المثـال الأول جميـع الأعـداد الصـحيحة الأصـغر من 100، كمـا تـرى في مقتطـف الشـيفرة البرمجيـة، ودون إضـافة قيـد، وسـينتهي بـك الأمـر بـرمي الاسـتثناء OutOfMemoryError إذا كنت تـرغب في بنـاء سلسـلة نصيّة تسـرد جميـع

الأرقام في المتتالية. سيُنشئ التابع `joinToString` نسخة من `StringBuffer` وسيستمر إضافة العناصر إليها، إذ أنَّه لا يوجد حد أعلى لها، وسيؤدي ذلك في نهاية المطاف إلى امتلاء مساحة الكومة وقت التشغيل وسيُرمَي خطأ، وستحدث نفس النتيجة إذا حوَّلت النوع إلى قائمة أو طقم، ولذلك، تجنَّب تخزين متتالية غير محدودة في تجميعة. نترك جانبًا تفاصيل جامع المهملات (garbage collector) لجافا وتنفيذاته المختلفة، إذ تُستأنَف (resumed) بصمة الذاكرة (memory footprint) لتوابع `genearateSequence` هذه لمعامل النوع T الذي ستُرجعه. الجزء الأخير من المثال السابق يُنشِئ متتالية باستخدام بذرة بداية (starting seed) لإرجاع جميع الأعداد الفردية من 100 إلى 110.

هنالك اختلاف جوهري بين التوابع `generateSequence` المُوسَّعة المُحمَّلة تحميلًا زائدًا (overloaded)، فالمرور مرَّة ثانية على متغيِّر من النوع `Secuence` سينتهي برمي الاستثناء `IllegalStateException`، وسبب ذلك يكمن في تنفيذ الشيفرة البرمجية لـ `generateSequence(nextFunction:()->T?)`:

```
public fun <T : Any> generateSequence(nextFunction: () ->T?):  Sequence<T>
{
        return GeneratorSequence(nextFunction,
{ nextFunction() }).constrainOnce()
     }


     public fun <T> Sequence<T>.constrainOnce(): Sequence<T> {
        return if (this is ConstrainedOnceSequence<T>) this else
ConstrainedOnceSequence(this)
}


     private class ConstrainedOnceSequence<T>(sequence: Sequence<T>) :
Sequence<T> {
        private val sequenceRef =
java.util.concurrent.atomic.AtomicReference(sequence)

        override fun iterator(): Iterator<T> {
           val sequence = sequenceRef.getAndSet(null) ?: throw
```

```
IllegalStateException("This sequence can be consumed only once.")
          return sequence.iterator()
        }
     }
```

في هـــذه الحالـــة وكمـــا تـــرى في المقتطـــف الســـابق، ستســـلمك المكتبـــة القياســـية نســـخة من ConstrainedOnceSequence. فـأي محاولـة لتطـبيق تكـرار (iterate) مـرة ثانيـة تنتهي بـترمي الاسـتثناء المذكور سابقًا، إن توثيق الواجهة البرمجية API جيّد للغاية، وسيُعلِمك إذا كـانت النتيجـة هي متتاليـة يمكن المُرور عليها مرّة واحد فقط أم أكثر.

دعنا نرى كيف يمكننا استخدام واجهة المتتالية البرمجية (API sequence) لقراءة ملف، وستبحث الشيفرة البرمجيّة التالية على ملف مورد (resource file)، وستستخدم مكتبة Java I/O لفتحـه وقراءتـه سـطرًا بسـطر حتى يتم إرجاع قيمة فارغة null:

```
val stream =
Thread.currentThread().javaClass.getResourceAsStream("/afile.txt")
val br = BufferedReader(InputStreamReader(stream))
val fileContent = generateSequence { br.readLine() }.takeWhile {  it !=
null }
println("File content:${fileContent.joinToString(" ")}")
```

من أجـل البسـاطة، تخلينـا عن الشـيفرة البرمجيـة الـتي تعـالج الأخطـاء، وسـتكون مخرجـات هـذه الشـيفرة البرمجية:

```
Kotlin is awesome!
```

عند التحدث عن المتتاليات، فإن توليد متتالية فيبوناتشي هـو أمـر لابـد منـه بالنسـبة للبعض، لـذلك، دعنـا نتبـع العادة ونرى كيف يمكنك كتابة ذلك في كوتلن:

```
var prevNumber: Int = 0
     val fibonacci1 = generateSequence(1) {
```

```
            val tmp = prevNumber
            prevNumber = it
            it + tmp
        }
        println("Fibonacci sequence: $
{fibonacci1.take(12).joinToString(",")}")
```

تتبع متتالية فيبوناتشي القاعدة التالية: أي عدد بعد أول عددين هو مجموع العددين السابقين: 1,1,2,3,5,8,13,21,34,55,89,144 وهكذا. في التشغيل الأول لتنفيذ المتتالية، حافظنا على مرجع للأرقام السابقة، نفِّذ الشيفرة السابقة وسترى الأعداد المذكورة آنفًا. وهذا التنفيذ مع ذلك ليس مثاليًا، فهل يمكننا إنشاء متتالية دون الاضطرار إلى إغلاق المتغيّر `prevNumber`؟ الظاهر أنَّنا نستطيع ذلك وإليك الشيفرة:

```
val fibonacc2 = generateSequence(1 to 1) {
            it.second to it.first + it.second
        }.map { it.first }
        println("Fibonacci sequence: $
{fibonacc2.take(12).joinToString(",")}")
```

المحاولة الثانية أكثر تعقيدًا بعض الشيء، إذ ستنتج متتالية من الأزواج عدد صحيح-عدد صحيح التي تحتوي على عدد فيبوناتشي الحالي والتالي مثل 1-1 و 1-2 و 2-3 و 3-5 و 5-8 و 8-13 وهكذا دواليك؛ ومن ثم، عن طريق الدالة `map`، فإنَّها تحدِّد أول عنصر من كل زوج، للحصول على نفس نتيجة المثال السابق.

## 9. خلاصة الفصل

لقد رأينا كيف نستخدم واجهة برمجة تطبيقات التجميعات بالتفصيل، ولقد تعلّمت كيف توفِّر مكتبة كوتلن القياسيّة أنواع تجميعات قابلة للتغيير وغير قابلة للتغيير، وكيف تُحقِّق عدم قابلية التغيير على مستوى الواجهة. أنت تعلم الآن أن كوتلن لا تضيف أية تجميعة جديدة، ولكن تعتمد بالأحرى على مكتبة تجمعات جافا الكبيرة، والتسمية المستعارة (aliasing) التي يقوم بها المصرِّف لم تعد لغزًا بعد اليوم. يمكنك الآن الذهاب واستخدام المصفوفات استخدامًا صحيحًا لأنك تميل إلى استخدام تنفيذات محدَّدة على المستوى العام عندما يتعلَّق الأمر بالأنواع الأساسيَّة.

توفِّر لك مكتبة كوتلن القياسيّة وحدات بناء للتعبير عن الحسابات المُعقَّدة عبر بضعة توابع مُوسّعة، ونأمل أن قدمنا لك منظورًا مختلفًا عندما يتعلَّق الأمر باختيار لغة مشروعك القادم.

يغطي الفصل التالي اختبار الوحدة والتكامل باستخدام كوتلن وذلك باستخدام أطر اختبار الوحدة (unit test frameworks).

الفصل الحادي عشر:

# الاختبار في كوتلن



من الأشياء الأولى التي يفعلها المطورون في الغالب عند تبني أو تقييم لغة جديدة هي تجربتها تدريجيًا، بدءًا من اختبارات الوحدة `Unit test`. ميِّزة استخدام هذا النهج هو أنَّه بما أن الاختبارات الخاصة بك لن تدخل في الإنتاج، فإن أي مشاكل مع اللغة أو أخطاء في مكتبة اللغة لن تؤثر على الشيفرة البرمجيّة الحقيقية، فهي تعطي فرصة للمطورين لتقييم ما إذا كانت اللغة مناسبة لاحتياجاتهم دون القلق بشأن الحاجة إلى إعادة كتابة الأجزاء الحرجة من شيفرتهم البرمجيّة إذا قرروا رفض اللغة الجديدة.

في هذا الفصل، سنقدّم مكتبة اختبار قويَّة في كوتلن وهي Kotest (كانت تعرف سابقًا باسم KotlinTest)، هذه المكتبة مفتوحة المصدر ومتاحة على GitHub، ومن خلال الاستفادة من مميزات كوتلن القويَّة، فهي توفِّر ميزات اختبار تفوق أطر اختبارات جافا النموذجيَّة مثل Junit أو TestNG المقدَّمة حاليًا.

| تنبيه | هذا الفصل يستعمل الإصدار 2.0 من مكتبة Kotest والتي كانت تُعرَف آنذاك باسم KotlinTest أي أن الشيفرات والمعلومات الواردة في هذا الفصل بنيت على الإصدار 2.0 والإصدار الحالي من المكتبة أثناء مراجعة الكتاب هي 4.0، لذا تأكد من قراءتك لأحدث نسخة منشورة من هذا الكتاب التي سيضاف إليها تحديثًا لهذه المكتبة. |
|---|---|

## 1. البداية

إن كتابة اختبارك الأول باستخدام KotlinTest هو أمر بسيط للغاية، فأولًا، تحتاج إلى إضافة تبعيَّة للبناء الخاص بك، فأسهل طريقة للقيام بذلك إذا كنت تستخدم Gradle أو Maven هو البحث في Maven central عن `io.KotlinTest`، زر http://search.maven.org واحصل على أحدث نسخة، ستحتاج إلى إضافتها إلى بناء Gradle باستخدام التالي:

```
testCompile 'io.KotlinTest:KotlinTest:2.0.0'
```

وبدلًا من ذلك، بالنسبة إلى Maven، هو استخدام الشيفرة البرمجيّة التالية:

```
<dependency>
   <groupId>io.KotlinTest</groupId>
   <artifactId>KotlinTest</artifactId>
```

```
    <version>2.0.0</version>
    <scope>test</scope>
</dependency>
```

تأكد من استعمال أحدث إصدار من المكتبة وتعديل ما يقابل ذلك في الشيفرة منها رقم إصدار المكتبة، وإن واجهك أي خطأ، ارجع إلى توثيق المكتبة وسجل التغييرات.

بعد ذلك، أنشئ مجلد مصدر الاختبار (في العادة `src/test/kotlin`) إذا لم يكن موجودًا بالفعل. سنكتب وحدة اختبار لصنف `String` للمكتبة القياسيّة، لذلك أنشئ ملف يدعى StringTest.kt وبداخل هذا الملف، أنشئ صنفًا مفردًا يدعى `StringTest` والذي سيوسع `FunSpec`، يجب أن تكون محتويات هذا الملف كما يلي:

```
import io.KotlinTest.specs.FunSpec
    class StringTest : FunSpec()
```

لكتابة وحدة اختبار، فإننا نستدعي دالة تدعى `test`، والتي تأخذ معاملين، الأول هو وصف الاختبار، والثاني هي دالة مُجرَّدة تحتوي على جسد الاختبار. سيظهر وصف أو اسم الاختبار في المخرجات حتى نعرف الاختبارات التي فشلت والتي نجحت.

بالنسبة لاختبارنا الأول، سنؤكد على ضرورة أن دالة `startsWith` المعرّفة في `String` يجب أن تُرجع القيمة `true` للبادئات الصحيحة، ضع كل اختبار فريد داخل كتلة `init {}` في جسد الصنف:

```
class StringTest : FunSpec() {
        init {
          test("String.startsWith should be true for a prefix") {
            "helloworld".startsWith("hello") shouldBe true
          }
        }
    }
```

لاحظ استخدام `shouldBe true`، هذه دالة `infix` التي تقبل قيمة وتُجري اختبار المساواة، إذا لم تتطابق القيم، فسيفشل الاختبار. تشير KotlinTest إلى مثل هذه

الدوال بالتأكيدات.

## 2. اختيار الأنماط

في الاختبار الأول الذي كتبناه، وسعنا صنفًا يدعى FunSpec والذي هو مثال على ما تسميه KotlinTest وصف spec، أو نمط style، فالوصف أو النمط هي الطريقة التي توضع فيها الاختبارات في ملفات الصنف. هنالك العديد من الأنماط المختلفة المتاحة، ويمكنك اختيار أي واحدة لاستخدامها. إن صنف FunSpec هو النمط الأكثر مماثلة إلى النمط Junit القديم، والذي قد يكون مألوفًا لك إن كنت تملك خلفية عن جافا.

سيغطي الجزء المتبقي من هذا القسم الأنماط المختلفة المتاحة لك للاختيار من بينها. النمط البديل الأول لصنف FunSpec هو FlatSpec ويفرض هذا على المستخدم استخدام كلمة should في أسماء الاختبارات، وقد يثير هذا إعجاب المطورين الذين يحبون توحيد أسماء الاختبارات:

```
class MyTests : FlatSpec() {
        init {
           "String.length" should "return the length of the string" {
              "hello".length shouldBe 5
              "".length shouldBe 0
           }
        }
     }
```

لدينا أيضًا WordSpec والذي يشبه FlatSpec إلى حد كبير، والذي يستخدم مرَّة أخرى كلمة should. ومع ذلك، بدلا من كتابة أسماء الاختبارات على نفس السطح (التسطيح)، فستجعلها متداخلة:

```
class MyTests : WordSpec() {
        init {

           "String.length" should {
              "return the length of the string" {
                 "hello".length shouldBe 5
```

```
                "".length shouldBe 0
            }
        }
    }
}
```

النمط التالي هو ShouldSpec، والذي هو تقريبًا نفس صنف FunSpec. هنالك فرق واحد هـو أن اسـم الدالـة هو should بدلا من test:

```
class MyTests : ShouldSpec() {
    init {
        should("return the length of the string") {
            "hello".length shouldBe 5
        "".length shouldBe 0
        }
    }
}
```

مع ShouldSpec، يمكن أن تكون الاختبارات متداخلة داخـل السلاسـل النصـيّة إذا كنت تـرغب في الحصـول على اختبارات متداخلة تتشارك في نفس فضاء اسم الأب (parent namespace)، وما نعنيه بفضـاء اسـم الأب هو أن تجمع هذه الاختبارات معًا في تسلسل هرمي داخل بيئة التطوير IDE:

```
class MyShouldSpec : ShouldSpec() {
    init {
        "String.length" {
            should("return the length of the string") {
                "hello".length shouldBe 5
            }
            should("support empty strings") {
                "".length shouldBe 0
            }
```

```
            }
        }
    }
```

وبعد ذلك، لدينا BehaviorSpec، ويهدف هذا إلى الأشخاص الذين يحبون بناء هيكل اختبارات بنمط المواصفات التي غالبًا ما يُنظر إليه في التطوير القائم على السلوك (behavior-driven development). الاختبارات متداخلة في الكتل الثلاثة تسمى given و when و then، عند الجمع، تقرأ هذه الكتل كفقرة مكتوبة بلغة إنجليزية عادية وليس لغة برمجة. بالإضافة إلى ذلك، عند النظر إلى تقرير الاختبارات المنفذة، فإنه يمكن قراءة نمط السلوك بشكل جيّد حتى لغير المطورين.

في كوتلن، تعد when كلمة مفتاحيّة، لذلك يجب عليك التوقف عن استخدامها مع علامات اقتباس مائلة ( ` backticks)، وبدلًا من ذلك يمكننا استخدام مكافئات عناوين الحالات الخاصة التي تسمى Given و When و Then المتوفّرة أيضًا:

```
class MyBehaviorSpec : BehaviorSpec() {
        init {
           given("a stack") {
              val stack = Stack<String>()
              `when`("an item is pushed") {
                 stack.push("kotlin")
                 then("the stack should not be empty") {
                    stack.isEmpty() shouldBe true
                 }
              }
              `when`("the stack is popped") {
                 stack.pop()
                 then("it should be empty") {
                    stack.isEmpty() shouldBe false
                 }
              }
```

```
            }
        }
    }
```

النمط التالي التي سنغطيه هو FeatureSpec والذي يشبه BehaviorSpec والفرق بينهما هو أنه يستخدم الكلمتين المفتاحيتين feature و scenario:

```
class MyFeatureSpec : FeatureSpec() {
        init {
            feature("a stack") {
                val stack = Stack<String>()
                scenario("should be non-empty when an item is pushed") {
                    stack.push("kotlin")
                    stack.isEmpty() shouldBe true
                }
                scenario("should be empty when the item is popped") {
                    stack.pop()
                    stack.isEmpty() shouldBe false
                }
            }
        }
    }
```

نمط المواصفات الأخير هو String، كما يوحي الاسم، فإنه يستخدم السلاسل النصية لجمع الاختبارات. هذا النمط هو الأبسط:

```
class MyStringSpec : StringSpec() {
        init {
            "strings.length should return size of string" {
                "hello".length shouldBe 5
            }
        }
```

```
    }
```

عندما لا تكون متأكدًا من المواصفات التي يجب عليك اختيارها، فاستخدم StringSpec، فهو منصوح به من مطوري KotlinTest.

## 3. المطابقات

يشار إلى اختبار المطابقات (Matchers test) لبعض الخاصيات باسم المطابق بعده مساواة بسيطة، فعلى سبيل المثال، قد يتحقق المُطابِق ما إذا كانت السلسلة النصية فارغة أو لا أو ما إذا كان العدد الصحيح موجب. في فقرة البداية السابقة، استخدمنا تأكيد shouldBe للتأكد من المساواة، وفي الواقع، يجب عليك التأكد أنَّ shouldBe تقبل مطابقةً يا وفِّر المزيد من تأكيدات المعقّدة.

الهدف من تسمية shouldBe هو الحصول على تأكيدات قابلة للقراءة مثل thisString shouldBe empty()، ولتعزيز هذا الهدف هنالك ما يعادل shouldBe وهو should، وباستخدامه يمكن كتابة مطابقات مثل thisString should startWith("foo") وستكون قابلة للقراءة كأنها لغة طبيعيَّة.

توفِّر KotlinTest العديد من المطابقات من خارج الصندوق، وكل واحد يحقق بعض الخاصيات أو شروط، في الجزء المتبقي من هذا القسم، سنغطي بعض المطابقات الأساسيّة.

### أ. مطابقات السلسلة النصيَّة

واحدة من التجميعات الأكثر شيوعًا من المطابقات هي مطابقات السلسلة النصية، وهذا ليس مفاجئًا، نظرًا لكمية استخدام السلسلة النصية الأساسية خلال تطوير البرمجيات، ويسرد الجدول التالي تطابقات السلسلة النصيّة الشائعة:

| مثال لمطابق | الوصف |
|---|---|
| hello world" should" startWith("he") | يختبر بداية السلسلة النصيّة |
| hello" should include("ell")" | يختبر السلسلة النصيّة الفرعية |

| | |
|---|---|
| `hello" should endWith("ello")"` | يختبر نهاية السلسلة النصيّة |
| `hello" should haveLength(5)"` | يختبر طول السلسلة النصيّة |
| `hello" should match("he...")"` | يختبر المطابقة باستخدام التعابير النمطية (regular expression) |

## ب. مطابقات التجميعات

تعمل المجموعة الأكثر فائدةً التالية من المطابقات على التجميعات، بما في ذلك القوائم والأطقم والخرائط وغيرها:

| مثال المطابق | الوصف |
|---|---|
| `col should contain(element)` | يختبر أن التجميعة يجب أن تحتوي على عنصر معيّن. |
| `col1 should haveSize(3)` | يختبر طول التجميعة. |
| `list shouldBe sorted<Int>()` | يختبر أن التجميعة يجب أن تكون مرتّبة، ويعمل هذا فقط للقوائم التي تحتوي على الأصناف المتفرعة من Comparable. |
| `col shouldBe singleElement(element)` | يختبر ما إذا كانت التجميعة تحتوي على عنصر واحدة فقط وهو العنصر الممرّر. |
| `col should containsAll(1, 2, 3)` | يختبر ما إذا كانت التجميعة تحتوي على هذه العناصر، الترتيب غير مهم. |
| `col should beEmpty()` | يختبر ما إذا كانت التجميعة فارغة أو لا. |
| `map should haveKey(key)` | يختبر ما إذا كانت خريطة تحتوي على مفتاح مرتبط لأي قيمة. |

| | |
|---|---|
| `map should haveValue(key)` | يختبر ما إذا كانت خريطة تحتوي على قيمة لمفتاح واحد على الأقل. |
| `map should contain(key, value)` | يختبر ما إذا كانت خريطة تحتوي على مفتاح معين لقيمة محددة. |

### ت. مطابقات الأعداد العشرية

من المطابقات المفيدة أيضًا مطابقات مجال التسامح (tolerance matcher)، والتي تعرَّف على الأعداد العشرية مضاعفة الدقة (doubles)، عند اختبار المساواة بين عدديّن عشريين، فلا يجب على المرء استخدام عملية المساواة البسيطة، وهذا بسبب طبيعة عدم دقة تخزين بعض القيم، وبشكل أساسي تكرار الأعداد الصحيحة بأساس 2 (مثلا الثلث لا يمكن تمثيله بالضبط بأساس 10).

أفضل وأأمن طريقة لإجراء الموازنة بين الأعداد العشرية هي تأكيد الفرق بين عددين عشريين أقل من قيمة معينة، وتدعى هذه القيمة بالقيمة المسموح عنها (tolerance)، ويجب أن تكون منخفضة بما فيه الكفاية لتلبية المعايير الخاصة بك للأعداد المطلوب التحقق من تساويها الدقيق، وتدعم KotlinTest هذا بشكل مدمج:

```
a shouldBe 1.0
a shouldBe (1.0 plusOrMinus 0.001)
```

يمكن أن يؤدي المثال الأول إلى أخطاء إذا كانت النتيجة المخزنة ليست 1,0 بالضبط، أما المثال الثاني فيستخدم عامل التسامح 0.001 وينفِّذ موازنة الفرق المطلق.

## توقع استثناءات

في بعض الأحيان، نريد التأكد من أنَّ الدالة سترمي استثناء، ربما لاختبار شرط مسبق الذي أضفناه، النهج الساذج لهذا سيكون تغليف استدعاء الدالة في كتلة `try...catch` ورمي الاستثناء في جزء `try`، وهذا مثال على دالة سترمي استثناء عند استدعاءها على عدد غير موجب:

```
fun squareRoot(k: Int): Int {
   require(k >= 0)
```

```
    return Math.sqrt(k.toDouble()).toInt()
}
```

وهذا هو نهجنا الأولي لاختباره:

```
try {
    squareRoot(-1)
    throw RuntimeException("This test should not pass")
} catch (e: Exception) {
    // noop
}
```

ومع ذلك، كما قد تكون خمنت على الأرجح، يمكن أن تعتني KotlinTest بهذا الأمر لنا، فهي توفِّر كتلة shouldThrow، والتي تتحقق من رمي الاستثناء، وإذا لا، سيفشل الاختبار بالنسبة لنا:

```
shouldThrow<IllegalArgumentException> {
    squareRoot(-1)
}
```

لاحظ أنَّ shouldThrow ستفشل في الاختبار إذا رُمِي نوع خطأ من الاستثناء، فنحن نتوقع

هنا IllegalArgumentException، ولذلك، على سبيل المثال، RuntimeException عامة قد تتسبب في فشل الاختبار.

## ث. الجمع بين المطابقات

يمكن جمع المطابقات معًا باستخدام العوامل المنطقية (Boolean logical operators) الاعتيادية and و or، ويمكننا القيام بذلك باستخدام دوال infix التي سميت للمعاملات:

```
val thisString = "hello world"
thisString should (haveLength(11) and include("llo wor"))
```

في المثال السابق، يجب اجتياز كلا المطابقين أو سيفشل الاختبار، لاحظ الأقواس حول المطابقات، ويجب علينا استخدام الكلمات المفتاحية should أو shouldBe مرَّة واحدة فقط:

```
val thisString = "hello world"
thisString should (haveLength(11) or include("goodbye"))
```

في المثال الثاني، يجب اجتياز اختبار واحد فقط، وتقيِّم المتطابقات بتكاسل، لذلك، إذا نجح المتطابق الأول، فلن يُستدعَى الثاني.

## ج. مطابقات مخصَّصة

رأيت الآن استخدام بعض المطابقات المضمنة التي توفرها KotlinTest، لكن لا تحتاج إلى التوقف عند هذا الحد، فتدعم KotlinTest، كتابة مطابقاتك المخصصة، وهذا الأمر سهل للغاية.

كل مطابق هو مجرَّد نسخة لواجهة Matcher:

```
interface Matcher<T> {
  fun test(value: T): Result
}
```

يجب على كل مطابق أن ينفِّذ دالة اختبار واحدة من شأنها أن تقبل قيمة كمعامل على الجانب الأيسر من دالة shouldBe. القيم على الجانب الأيمن من دالة shouldBe هي مجرّد معاملات باني (constructor parameter) للمطابق الممرَّر عبر الدالة المسؤولة عن إنشاء المطابق. يجب على دالة الاختبار أن تُرجع نسخة Result، والتي تحتوي على راية منطقية (Boolean flag) لمعرفة ما إذا نجح الاختبار أو لا وسيُرجع رسالة إذا لم ينجح.

لأغراض العرض، سنحاول تحسين تجربة ملفات الاختبار، لنفترض أننا نكتب وحدة اختبار ونرغب في تجربة أن الملف موجود وأن الملف هو ملف صورة. بالنسبة لمثال المطابق المخصص، يكفي أن نفترض أن الملف هو صورة إذا كان لديه صيغة صورة معروفة، في الواقع، قد ترغب في الذهاب أبعد من ذلك عبر محاولة تحميل محتويات الملف لتأكيد إذا كان صورة بالفعل.

الخطوة الأولى هي إنشاء دالة ستعيد مطابقًا، الدالة هي ماذا سيرى المستخدمون على الجانب الأيمن من

shouldBe، لذلك من الأفضل أن يكون له اسم يُقرأ بشكل صحيح:

```
fun anImageFile() = object : Matcher<File> {
      override fun test(value: File): Result {
      }
```

لاحظ أن المطابق يملك معامل نوع، والذي هو نوع قيمة المطابق التي يمكن استخدامها للتحقق، إن الدالة shouldBe هي دالة تابعة معرَّفة على معامل النوع هذا، وبالتالي فإن المصرِّف سيحد من استخدام المطابقة لهذه الأنواع فقط، وبعبارة أخرى، فإن مطابق الملفات لدينا سيعمل على متغيرات الملف، ولن تكون قادرًا على استخدامه على سلسلة نصيّة عن طريق الخطأ.

إن تنفيذ مطابق مخصص بسيط للغاية، نحن بحاجة إلى استخدام الدالة الموجودة على كائن File والتحقق من الاسم:

```
val anImageFile = object : Matcher<File> {
   private val suffixes = setOf("jpeg", "jpg", "png", "gif")
   override fun test(value: File): Result {
      val fileExists = value.exists()
      val hasImageSuffix = suffixes.any {
      value.name.toLowerCase().endsWith(it) }
      if (fileExists.not()) {
         return Result(false, "File $value should exist")
      }
      if (!hasImageSuffix) {
         return Result(false, "File $value should have a well known
         image suffix")
      }
      return Result(true, "Test passed")
   }
}
```

بمجرَّد تنفيذ المطابق، كل ما يتبقى هو أن تستخدمه، يجب استدعاء الدالة على نطاق، لذلك يجب أن تكون إما

دالة ذات مستوى عالي مع الاستدعاء أو يجب وضعها على نوع أعلى (supertype) ووراثتها:

```
class MatcherTest : FunSpec() {
   init {
      test("testing our file matcher") {
         val file = File("/home/packt/kotlin.jpg")
         file shouldBe anImageFile
      }
   }
}
```

لاحظ كيف يقرأ التأكيد الآن كبيان صحيح نحويًا، وعلى الرغم من أنَّه لا توجد متطلبات على اسم الدوال بهـذه الطريقة، فهذا يعني أنَّها ستجعل الاختبار أسهل للقراءة بالنسبة لشخص غير مطلع على الشيفرة البرمجيَّة.

من خلال كتابة مطابقات إلى واجهة `Matcher`، فيمكن استخدامها تلقائيًا في العوامل المنطقيـة. لنتخيـل أننـا نرغب في جلب الوظائف exists في مطابق منفصل ومن ثم نقدّم مطابقًا للاختبار على نوع ملف معيَّن:

```
fun exist() = object : Matcher<File> {
   override fun test(value: File): Result {
      val fileExists = value.exists()
      return if (!fileExists) {
         return Result(false, "File $value should exist")
      } else {
         Result(true, "Test passed")
      }
   }
}
```

أضف الآن مطابق لاختبار ملف يحتوي على صيغة ملف معيَّن:

```
fun ofFileType(ext: String) = object : Matcher<File> {
   override fun test(value: File): Result {
      val isOfType = value.name.toLowerCase().endsWith(ext)
```

```
            return if (!isOfType) {
                Result(false, "File $value is not of type $ext")
            } else {
                Result(true, "Test passed")
            }
        }
    }
```

ثم استخدم هذين المطابقين معا باستخدام العاملين and/or العاديّة:

```
class MultipleMatcherTest : FunSpec() {
    init {
        test("testing our file matcher") {
            val dir = File("/home/packt/images")
            for (file in dir.listFiles()) {
                (file should exist()) and (file shouldBe
                ofFileType("jpeg"))
            }
        }
    }
}
```

يوضح هذا مدى السرعة التي يمكنك إضافة مطابقات مخصصة لتغليف منطق الاختبار، مما يجعلها قابلة لإعادة الاستخدام عبر عدة ملفات اختبار.

## 4. المفتشون

يعد مفتشو KotlinTest (أي inspectors) طريقة سهلة لاختبار محتويات التجميعات، فقد تود في بعض الأحيان التأكيد على أن بعض عناصر المجموعة فقط يجب أن تجتاز التأكيد، وفي بعض الأحيان الأخرى، قد لا تحتاج إلى عناصر لاجتياز التأكيد، فقط واحد أو اثنين وهكذا، وبالطبع، يمكننا القيام بذلك بأنفسنا فقط بالتكرار على التجميعة وتتبع كم من العناصر التي مرّت من التأكيد، ومع ذلك، يفعل المفتشون ذلك لنا.

لنبدأ بالحالة المعتادة التي نريد أن تمر جميع عناصر المجموعة بها التأكيدات، ولهذا، أولًا، قبل كل شيء، سنعرّف قائمة سنعمل معها خلال بقيّة القسم:

```
val kings = listOf("Stephen I", "Henry I", "Henry II", "Henry III",
      "William I", "William II")
```

وبعد ذلك، سنتأكد من أن كل ملك `king` يملك رقمًا ملكيًا ينتهي بحرف `'I'`، كالتالي:

```
class InspectorTests : StringSpec() {
   init {
"all kings should have a regal number" {
        forAll(kings) {
           it should endWith("I")
        }
      }
   }
}
```

كان يمكن أن يتحقق هذا الاختبار أيضًا مع دالة `All` على التجميعات. والحالات الأخرى ليست سهلة دون المفتشين، والمثال التالي سيظهر ذلك:

```
class InspectorTests : StringSpec() {
   init {
      "only one king has the name Stephen" {
        forOne(kings) {
           it should startWith("Stephen")
        }
      }
   }
}
```

بدون المفتش، سيكون علينا أن نلتقط الاستثناءات ونحسب عدد الملوك الذين اجتازوا الاختبار، ويخفي المفتش هذا الجزء لنا. المفتش التالي سيكون مثيرًا للاهتمام:

```
class InspectorTests : StringSpec() {
  init {
    "some kings have regal number II" {
      forSome(kings) {
        it should endWith("II")
      }
    }
  }
}
```

هذا مثال على مفتش forSome الذي يؤكد أن عنصر واحد على الأقل، وليس جميع العناصر، قد اجتاز الاختبار، ولذلك بالنسبة لـ n عناصر في التجميعة، فإن الاختبار سيمر إذا تطابق عنصر بين 1 و n-1 مع التأكيد.

هنالك العديد من المفتشين، لكن سنعرض واحد آخر فقط:

```
class InspectorTests : StringSpec() {
  init {
    "at least one King has the name Henry" {
      forAtLeastOne(kings) {
        it should startWith("Henry")
      }
    }
  }
}
```

مفتش forAtLeastOne، كما يوحي الاسم، يتحقق ببساطة من وجود عنصر واحد قد اجتاز الاختبار، وهو يختلف عن مفتش forSome ولذلك سيسمح لجميع العناصر أن تمر.

## 5. المعترضات

عند التحرك خارج نطاق اختبارات الوحدة المستقلة ونحو الاختبارات التي تتطلّب مواردًا، فسنحتاج في

الكثير من الأحيان إلى إعداد هذه الموارد قبل الاختبار وتدميرهم لاحقًا، على سبيل المثال، قد يلزم تهيئة اتصال قاعدة البيانات لاستخدامها من قبل الاختبار ومن ثم إغلاقها بشكل صحيح بمجرّد الانتهاء من الاختبار؛ يمكننا فعل ذلك يدويًا في الاختبار، ولكن إذا كان لدينا مجموعة من الاختبارات، فسيصبح هذا الأمر شاقًا.

ألن يكون ذلك أجمل إذا كان بإمكاننا فقط تحديد دالة مرّة واحدة ومن ثم تشغيلها قبل وبعد كل اختبار أو كل مجموعة من الاختبارات، هذه الوظيفة موجودة في KotlinTest تحت اسم المعترضات (interceptors). يعرَّف كل نوع من المعترضات للتشغيل قبل وبعد اختبار الشيفرة البرمجيّة، دعنا نناقش أنواعًا مختلفة من المعترضات.

### أ. معترض حالة الاختبار

النوع الأول من المعترضات هو معترض حالة الاختبار (test case interceptor)، تضاف هذه المعترضات مباشرةً إلى حالات الاختبار نفسها، وهي تنطبق فقط على حالات الاختبار التي أضيفت إليها. يتلقى معترض حالة الاختبار معاملين، الأول هو سياق حالة الاختبار، ويحتوي هذا على تفاصيل الاختبار، مثل اسم الاختبار، ومكان ملف المواصفات وكم عدد الاستدعاءات التي يجب أن تكون، وهكذا. المعامل الثاني هو الاختبار نفسه في شكل دالة عديمة الرتبية (zero arity). يجب استدعاء هذه الدالة من قبل المعترض أو سيتخطى الاختبار. سيعطي هذا معترضات الاختبار القدرة على اختيار ما إذا كان سيجري الاختبار أم لا.

في المثال التالي، سنعرِّف معترضًا سيرجع لنا الوقت الذي يستغرقه إجراء الاختبار:

```
val myinterceptor: (TestCaseContext, () -> Unit) -> Unit = {
  context, test ->
  val start = System.currentTimeMillis()
  test()
  val end = System.currentTimeMillis()
  val duration = end - start
  println("This test took $duration millis")
}
```

لاحظ كيف يُستدعى الاختبار داخل المعترض، الخطوة التالية هي إضافة المعترض إلى الاختبارات التي تريد توقيتها:

```
"this test has an interceptor" {
   // test logic here
}.config(interceptors = listOf(myinterceptor))
"so does this test" {
// test logic here
}.config(interceptors = listOf(myinterceptor))
```

لاحظ أن كل حالة اختبار تقبل قائمة من المعترضات، على الرغم من أننا في هذه الحالة استخدمنا واحدة فقط، فيمكننا إضافة رقم عشوائي.

### ب. معترض المواصفات

النوع التالي من المعترضات هو معترض المواصفات (spec interceptor)، ويُستخدم لاعتراض جميع الاختبارات في صنف اختبار واحد، يتشابه معترض المواصفات إلى معترض حالة الاختبار، الفرق الوحيد هو أن سياق حالة الاختبار يُستبدَل بسياق المواصفات، ومثل المعترض السابق، يجب استدعاء الدالة المقدمة، وخلاف لذلك، سيتخطى كامل المواصفات، لذلك يمنحك ذلك القدرة على استخدام المنطق المخصَّص لتحديد ما إذا كانت المواصفات ستشتغل أو لا:

```
val mySpecInterceptor: (Spec, () -> Unit) -> Unit = {
   spec, tests ->
   val start = System.currentTimeMillis()
   tests()
   val end = System.currentTimeMillis()
   val duration = end - start
   println("The spec took $duration millis")
}
```

قمنا هنا بتنفيذ معترض الوقت مرَّة أخرى، وهذه المرَّة لوقت المواصفات الكاملة. لاستخدام هذا، تجاوزنا خاصيَّة تسمى `specInterceptors` توفِّر قائمة من المعترضات:

```
override val specInterceptors: List<(Spec, () -> Unit) -> Unit> =
listOf(mySpecInterceptor)
```

كل هذا مشابه للغاية لمثال حالة الاختبار.

## 6. ضبط المشروع وتهيئته

في بعض الأحيان قد ترغب في تنفيذ بعض التعليمات البرمجيّة قبل إجراء أي اختبارات على الإطلاق أو بعد اكتمال جميع الاختبارات (سواء كانت ناجحة أم لا). يمكن تحقيق ذلك من خلال استخدام صنف المجرّد `ProjectConfig` وذلك من خلال إنشاء كائن يوسّع هذا الصنف المجرَّد ويتأكد من أنها على مسار الصنف. وبعد ذلك، سيجد KotlinTest تلقائيًا وسيستدعيها:

```
object MyProjectConfig : ProjectConfig() {
   var server: HttpServer? = null
   override fun beforeAll() {
      val addr = InetSocketAddress(8080)
      val server = HttpServer.create(addr, 0)
      server.executor = Executors.newCachedThreadPool()
      server.start()
      println("Server is listening on port 8080")
   }
   override fun afterAll() {
      server!!.stop(0)
   }
}
```

في هذه الحالة، أنشأنا نسخة `ProjectConfig` التي تنشئ خادم HTTP مُضمَّن بحيث يمكن لجميع الاختبارات استخدام هذا الخادم دون الحاجة إلى إنشاء خادم خاص بهم. بعد إكمال جميع الاختبارات، سيُغلَق الخادم مرّة أخرى.

لن نتمكن من وضع هذه الشيفرة البرمجيّة في أي مواصفات محدَّدة (مجموعة اختبارات) لأننا لا نعرف الترتيب الذي ستنفّذ به الاختبارات. إذن كيف نعرف أي ملف سيحتوي عليه؟ حتى لو كان هنالك أمر حتميًا، لا يوجد شيء يمكن القيام به لوقف مواصفات أخرى من إضافتها، أيهما يأتي أولًا في الطلب.

## 7. اختبار الخاصيّة

هنالك طريقة بديلة للاختبار شائعة في الأطر مثل `QuickCheck` في هاسكل و `ScalaCheck` في سكالا، وهي فكرة اختبار الخاصيَّة (property testing) والذي يهدف إلى اختبار خاصيَّة واحدة من الدالة في المرة الواحدة، فعلى سبيل المثال، عند تسلسل سلسلتين نصيتين مع بعضها البعض، يجب أن تظل خاصيَّة الطول ثابتة كمجموع قيمة الأطوال الأصلية، وهذا على نقيض النمط العادي للاختبار، والذي يعتمد على المثال.

| ملاحظة | لاحظ أن الخاصية في هذه الحالة لا تشير إلى خاصيات الكائنات، كما هو الحال في الحقول أو الأعضاء. بدلًا من ذلك، فإنها تشير إلى بعض الثوابت (invariant) أو الخبريات (predicate) التي يجب أن تكون محققة `true`. |
|---|---|

بالنظر إلى أننا سنقوم باختبار أن خاصيَّة تحمل قيم إدخال متعددة، فيتبع ذلك أنَّنا نرغب في استخدام أكبر عدد ممكن من القيم المختلفة، ولهذا، فإن الاختبار المبني على الخاصيّة غالبًا ما يرتبط مع التوليد التلقائي لقيم الإدخال. توفِّر KotlinTest هذه القيم من خلال المولدات المسماة باسم مناسب (aptly named generators).

لاستخدام مولد لاختبار الخاصيَّة، نحتاج إلى استخدام استدعاء نمط المفتش والذي سنمرر دالة اختبار أخرى إليه. يجب أن تملك دالة الاختبار أنواع المعامل المحددّة لأن المصرّف لن يكون قادرًا على استنتاجها. يجب على نفس الدالة إرجاع قيمة منطقية (Boolean)، وستشير هذه القيمة المنطقية ما إذا كانت الخاصيّة مقتصرة على قيم الإدخال، أي أننا لا نحتاج إلى استخدام مطابق في اختبار الخاصيّة. قيمة الإرجاع للدالة ستشير في حد ذاتها إلى صحة الخاصيّة.

في هذا المثال، سنختبر طول السلسلة المتسلسلة، كما ذكرنا سابقًا، على سبيل المثال:

```
"String.size" {
  forAll({ a: String, b: String ->
    (a + b).length == a.length + b.length
  })
```

```
}
```

لاحظ أنَّ المعاملات لها صفة كتابية (ascriptions). تستخدم KotlinTest أنواع المعامل لتحديد نوع المولِّد لاستخدامه لكل مدخل. يتلقى كل مدخل نسخة خاص به من المولِّد حتى لو كانت الأنواع نفسها. استخدمنا `forAll` في مثالنا هذا، لكن يمكن أن نستخدمها بدلًا من ذلك، والذي هو العكس.

عند تنفيذ حالة الاختبار، سيعيد إطار العمل هذا الاختبار مئات المرات، في كل مرَّة تتطلَّب قيم جديدة من المولدات. لن تكون المدخلات بين a-z أو أي أبْجَعدديَّة (alphanumeric، وهو جمع بين أبجدية وعددية)، لكن يمكن أن تكون أي قيمة أو قيم `unicode`. هذا النوع من المولدات مفيد لتوليد مدخلات قد نتغاضى عنها عند كتابة اختبار وحدة على أساس المثال.

الفائدة الواضحة لهذا النهج هو أنَّه يمكننا اختبار العديد من التجميعات الأخرى يدويًا، وسيعطينا هذا ثقة أكبر في متانة شيفرتنا البرمجيّة. الجانب السلبي هو أنَّنا لا نسيطر على قيم محدَّدة، فقد تنزلق حالة حافة (edge case) عدَّة مرات قبل أن تُختَار كقيمة عشوائيَّة.

### أ. تحديد مولِّد

كما رأينا، غالبًا ما يُستخدم اختبار الخاصيّة لاختبار مدخلات متعدّدة بسرعة. ومع ذلك، في بعض الأحيان، قد لا تكون المولدات المتوفرة تلقائيًا هي بالضبط ما نريده، على سبيل المثال، إذا أردنا تجربة دالة الجذر التربيعي، فلن نرغب في توليد الأرقام السالبة، لذلك بدلًا من السماح لـ KotlinTest باختيار مولِّد افتراضي، سنقدم واحدًا يدويًا عند كتابة الاختبار:

```
"squareRoot" {
   forAll(Gen.int(), { k ->
      val square = squareRoot(k)
      square * square == k
   })
}
```

تأتي KotlinTest مع العديد من المولدات المضمنة، مثل الأعداد الطبيعية، الأعداد السلبية، الملفات

العشوائيَّة، وما إلى ذلك.

## مولِّد مُخصَّص

في بعض الأحيان، نريد تحديد مجالات أو قيم المدخلات الخاصة بنا بالكامل والمولدات المدمجة ليست كافية بما فيه الكفاية، وتكمن هنا فائدة المولدات المخصصة. تملك KotlinTest عدة طرائق لإنشاء مولِّد بشكل ملائم، على سبيل المثال، يمكننا إنشاء مولِّد يختار عنصرًا عشوائيًا من تجميعة ويعيده في كل مرة يُستدعَى فيها:

```
val values = listOf("pick", "one", "of", "these")
forAll(Gen.oneOf(values), { element ->
   // test logic
})
```

أو يمكننا إنشاء مولّد على مجال من الأعداد ونجعل المولّد يختار عددًا عشوائيًا منها:

```
forAll(Gen.choose(1, 10000), { k ->
   // test logic
})
```

بدلًا من ذلك، إذا لم يكن المساعدون المدمجون (built-in helpers) كافيين، فيمكننا دائمًا إنشاء واحدة بأنفسنا من الصفر، كل ما نحتاج له هو توسيع الواجهة Generator<T>، إذ أنَّ T هي نوع الإرجاع، ومن ثم تنفيذ الدالة generate. تُستدعَى الدالة generate في كل مرَّة يتطلب الإطار عددًا آخر، لذلك يجب ألا يخزَّن في الذاكرة المؤقتة، وفي المثال التالي، يُرجِع المولِّد المخصَّص عددًا صحيحًا عشوائيًا في كل مرة:

```
fun evenInts() = object : Gen<Int> {
   override fun generate(): Int {
      while (true) {
         val next = Random.default.nextInt()
         if (next % 2 == 0)
         return next
      }
   }
```

```
}
```

بمجرَّد الحصول على العدد الصحيح، يمكنك وضعه في المكان الصحيح:

```
forAll(evenInts(), { k ->
   val square = squareRoot(k)
   square * square == k
})
```

## 8. الاختبار القائم على جدول

تتشابه فكرة الاختبارات القائمة على الجدول (table-driven tests) مع فكرة الاختبارات القائمة على الخاصيّة (property-based testing)، الفرق هنا أنَّه بدلًا من مولدات توفِّر أرقام عشوائيّة، تكون مجموعة قيم المدخلات محدَّدة يدويًا، والطريقة التي نفعل بها ذلك هي عن طريق إعلان بنية جدول يمكن ضمه إلى الاختبار أو تحميله من ملف.

النهج الأسهل هو كتابة الجدول ببساطة، وهذا يعمل بشكل جيِّد إذا كان لدينا مجالًا صغيرًا من قيم الإدخال أو حالات معيَّنة نريد اختبارها. على سبيل المثال، قد تكون لدينا دالة مع ثلاثة قيم منطقيَّة (Boolean) ونرغب في اختبار التركيبات. الخطوة الأولى هي تعريف الجدول الذي يحتوي على التجميعات التي نريد اختبارها:

```
val table = table(
   headers("a", "b", "c"),
   row(true, true, true),
   row(true, false, true),
   row(true, false, false)
)
```

لاحظ أنَّنا نستخدم الدالتين المساعدتين `headers` و `row`، فالأولى مهمة ولا تُستخدَم للإدخال لكن تستخدَم لتسمية القيم التي ستفشل عندما لا تنجح في الاختبار، بالنسبة لهذا الجدول، سنمرِّره على شكل كتل، والتي سوف تشبه دوال المفتش (inspector functions) مرَّة أخرى:

```
forAll(table) { a, b, c ->
   a shouldBe true
   if (b)
   c shouldBe true
}
```

**ملاحظة**: لاحظ أنَّه في الاختبار القائم على الجدول، لا تحتاج الدالة إلى إرجاع قيمة منطقيَّة، لذا على عكس اختبار المبني على الخاصيَّة، يجب علينا استخدام المطابقات العاديَّة.

كما ذكرنا سابقًا، تُستخدَم الترويسات (headers) للإبلاغ عن الأخطاء، إذا فشل صف معيَّن، فإن الناتج سيكون مشابه لما يلي:

```
Test failed for (x, 9), (y, 12), (z, 18) with error 225 did not equal 324
```

هنالك تنفيذات من أجل `row` و `header` التي تغطي تجميعات من النوع `tuple` تصل إلى 22 عنصر، وإذا لم يكن هذا كافيًا، فإنَّ الدالة المُختبرَة ربما تكون كبيرة جدًا ويمكنك تقسيمها.

## أ. اختبار شيفرة غير حتميَّة

عند اختبار شيفرة غير قطعيَّة (non-deterministic code)، مثل `futures` أو `sctors` أو مخازن البيانات الثابتة، فمن المفيد أن تتأكَّد أنه في مرحلة ما، من المتوقع أن ينجح الاختبار، حتى لو فشل الاختبار في البداية، ومن الطرائق الشائعة لتحقيق ذلك هي استخدام مزلاج العد التنازلي (countdown latch) وتكييف الشيفرة البرمجيّة تحت الاختبار لفتح المزلاج بمجرَّد اكتمال الشيفرة البرمجيّة:

```
val latch = CountDownLatch(1)
createFile("/home/davidcopperfield.txt", { latch.countDown() })
latch.await(1, TimeUnit.MINUTES)
// continue with test
```

في المثال السابق، سنكون قادرين على استخدام المزلاج في دالة `createFile` لأنَّها تقبل مستمعًا يُستدعَى عند إنشاء الملف.

| ملاحظة | لاحظ أنَّ مزلاج العد التنازلي هو نوع أساسي متزامن (concurrency primitive) يحظر أي خيط يستدعي `await` عليه، حتى إنَّهاء العد عدَّة مرات. |
|---|---|

لا تعمل هذه الخدعة إذا لم نتمكن من تغيير الشيفرة تحت الاختبار لقبول بعض الدوال أو رد نداء أو مستمع. في بعض الأحيان، نضطر للرجوع إلى النهج الساذج للأسف، وهو جعل الخيط ينام (sleep) لبعض الوقت:

```
createFile("/home/davidcopperfield.txt")
Thread.sleep(5000)
// continue with test
```

لهذا النهج إشكالية لأنَّ مهلة النوم يجب أن تكون كبيرة بما يكفي لكي لا نستيقظ مبكرًا ونفشل في البناء، ومن ناحية أخرى، إذا كانت القيمة كبيرة للغاية، فسينخفض إنتاجية الاختبار إذ سنضطر لانتظار انتهاء مدة النوم، على الرغم من أنَّنا قد نكون قادرين على الاستمرار في وقت سابق.

ما نريده حقًا هو جعل الاختبار ينتظر عندما يكون الاختبار `false` ومن ثم إنهاء الاختبار بمجرّد أن تنقلب إلى `true`. تنفِّذ KotlinTest هذه الخدعة عن طريق ميزة تسمى `Eventually`، والتي هي مستوحاة من دالة مماثلة في `ScalaTest`.

لاستخدام ميِّزة `eventually`، يجب علينا توسيع الواجهة `Eventually` التي تقدِّم هذه الوظيفة:

```
class EventuallyExample : WordSpec(), Eventually
```

ومن ثم، استدعي الدالة `eventually` مع تمرير المدة أولًا والدالة المُجرَّدة التي ستنفَّذ ثانيًا. ستنفَّذ الدالة بشكل متكرِّر حتى تنتهي المدة أو تنتهي الدالة المُجرَّدة بنجاح الاختبار. لنزور مثال إنشاء الملف السابق، فيمكن إعادة كتابته باستخدام التابع `eventually` كالتالي:

```
class FileCreateWithEventually : ShouldSpec(), Eventually {
   init {
      should("create file") {
         eventually(60.seconds) {
            createFile("/home/davidcopperfield.txt")
```

```
            }
        }
    }
}
```

**ملاحظة**: لاحظ أنَّه نظرًا لتقييم الدالة عدَّة مرات، فلا يجب الاعتماد على أي حالة تغيَّرت في التشغيل السابق.

## 9. الوسوم والشروط والتهيئة

في هذا القسم، سنغطي باختصار خيارات التهيئة المختلفة التي يمكن استخدامها للسيطرة على كيفيَّة تنفيذ الاختبارات وما هي الاختبارات التي تُنفَّذ.

### أ. التهيئة (Config)

توفر كل حالة اختبار الدالة `config`، والتي يمكن استخدامها لضبط تكوينات محدَّدة لهذا الاختبار، مثل الخيوط (threading)، والوسوم (tags) وما إذا تم تمكين الاختبار أم لا. فعلى سبيل المثال، يمكننا تغيير عدد مرات تنفيذ الاختبار:

```
class ConfigExample : ShouldSpec(), Eventually {
    init {
        should("run multiple times") {
            // test logic
        }.config(invocations = 5)
    }
}
```

عيّنا عدد الاستدعاءات إلى 5، هذا هو عدد المرات التي سيُنفَّذ فيها الاختبار نفسه كل مرَّة يُستدعَى فيه مرحلة اختبارات الوحدة. بالإضافة إلى عدد الاستدعاءات هنالك عدد الخيوط التي ستُستخدَم، وهي بشكل افتراضي تساوي 1:

```
should("run multiple times in multiple threads") {
   // test logic
}.config(invocations = 20, threads = 4)
```

في هذا المثال، سيُشغَّل الاختبار 20 مرَّة، لكن سيُستخدَم مجمَّعٌ لأربعة خيوط لتنفيذ الاختبارات، ويجب علينا عند استخدام هذا الخيار بالطبع التأكُّد من أنَّ الاختبارات آمنة الخيوط.

الخيار المفيد الآخر هو تعيين مهلة يمكن فيها إنهاء الخيط إذا استغرق وقتًا طويلًا جدًا لينتهي، ويمكن أن يملك كل اختبار إعدادات التكوين الخاصة به مستقلًا عن الآخرين.

## ب. الشروط

الشروط هي طريقة بسيطة لتمكين أو تعطيل اختبارٍ يعتمد على تقييم وقت التشغيل، تحتوي كتلة التهيئة على خاصيَّة `enabled`، والتي تُستدعَى قبل تنفيذ الاختبار من أجل معرفة ما إذا كان يجب تنفيذ الاختبار أو تخطيه (لا تتحمَّل أي منطق إضافي في الاعتراض).

أبسط حالة تتمثَّل في ضبط قيمةٍ إلى `false`:

```
should("be disabled") {
   // test logic
}.config(enabled = false)
```

القيمة الافتراضي، إن أزيلَت، هي القيمة `true`. بشكل عام، فإنَّ خيار إيقاف الاختبار بالكامل من خلال إجبار القيم أن تكون `false` يجب أن يُستخدَم باعتدال. فلربما يمكنك استعادة آخر بناء سليم (green build) أثناء التحقق من سبب فشل الاختبار.

يمكننا توسيع هذا لاستخدام عملية بحث وقت التشغيل عن طريق تعريف دالة بدلًا من وضع القيم في الشيفرة البرمجية. في الواقع، يمكن استخدام أي شيء يمثِّل تعبيرًا، على سبيل المثال، قد نقرِّر أنَّنا نرغب فقط في تنفيذ الاختبار على نظام متعدِّد النوى:

```
fun isMultiCore(): Boolean =
Runtime.getRuntime().availableProcessors() > 1
```

```
should("only run on multicore machines") {
   // test logic
}.config(enabled = isMultiCore())
```

يمكن كتابة الشروط لدعم حالات استخدام عديدة، مثل تحديد الاختبارات على أنظمة تشغيل معيَّنة أو آلات ذات أجهزة معيَّنة أو تشغيل الاختبارات في أوقات معيَّنة من اليوم.

## ت. الوسوم

على غرار الشروط، تسمح الوسوم (tags) بتجميع الاختبارات بحيث يمكن تمكينها أو تعطيلها وقت التشغيل، ويمكن أن تحتوي كل حالة اختبار على وسم خاص بها أو مجموعة وسوم (يمكن أن تكون واحدة أو أكثر)، من الممكن أيضا تركها دون وسم. عند وقت التشغيل، يمكن تعيين خاصيَّة النظام، والتي من شأنها أن تحدِّد الوسوم التي تُضمَّن أو تُستبعَد، مما يشير إلى أن الاختبارات التي تطابق المتطلبات ستُنفَّذ فقط.

الوسم (tag) هو مجرَّد كائن يُوسَّع من الصنف المجرَّد `Tag`، اسم الوسم هو اسم الصنف دون أي فضاء اسم للحزمة. فعلى سبيل المثال، يمكننا وضع وسم اختبار مع متطلبات قاعدة بيانات ونظام تشغيل معيَّن:

```
object ElasticSearch : Tag()
 object Windows : Tag()
 should("this test is tagged") {
   // test logic
 }.config(tags = setOf(ElasticSearch, Windows))
```

إذا كنت تستخدم `Gradle`، فيمكننا تنفيذ هذه الاختبارات فقط باستخدام الأمر التالي:

```
gradle test -DincludeTags=Windows,ElasticSearch
```

إذا أردت استبعاد أي اختبار يتطلَّب نظام التشغيل ويندوز لأنَّنا نرغب في تشغيل وظيفة بناء على لينكس، فيمكننا استخدام الأمر التالي:

```
gradle test -DexcludeTags=Windows
```

إذا ضبطنا قيمة كل من الخاصيتين `include` و `exclude`، فإنَّ الوسم الذي يطابق كلا المجموعتين هو الذي



سيُنفَّذ، وإذا حذفت خاصيات وقت التشغيل كليًا، فستُنفَّذ جميع الاختبارات، وهو الوضع العادي.

| ملاحظة | نظرًا لأنَّه يُستخدَم اسم بسيط للوسم، فإذا عرَّفت وسوم متعدِّدة مع نفس الاسم في حزم مختلفة فستظهر على أنَّها نفس الوسوم بالنسبة إلى KotlinTest. |
|---|---|

### نسخة واحد

في بعض الأحيان، قد ترغب في الحصول على نسخة جديدة لصنف اختبار لكل حالة اختبار التي تنفَّذ، ربما لديك بعض شيفرات البرمجيَّة للتهيئة من خارج كتلة `{} init` وترغب في إعادة تهيئتها لكل اختبار. الطريقة السهلة لفعل ذلك هي عن طريق إنشاء نسخة `KotlinTest` جديدة لنسخة الصنف. ولفعل ذلك، استبدل قيمة الخاصيَّة `oneInstancePerTest` واضبطها إلى القيمة `true`:

```
class OneInstanceOfTheseTests : ShouldSpec() {
   override val oneInstancePerTest = true
   init {
      // tests here
   }
}
```

### ث. الموارد

واحدة من مميزات `KotlinTest` هي القدر على إغلاق الموارد تلقائيًا مرَّة واحدة عند إنهاء جميع الاختبارات، وهذا اختصار لكتابة اعتراض وإغلاقها بنفسك، وهو مفيد إذا كان كل ما عليك القيام به هو التأكد من إغلاق بعض المقابض:

```
class ResourceExample : StringSpec() {
   val input = autoClose(javaClass.getResourceAsStream("data.csv"))
   init {
      "your test case" {
         // use input stream here
```

```
        }
    }
}
```

إن استخدامها واضح للغاية، فما عليك القيام به سوى تغليف المورد، مثل مجرى الدخل في هذا المثال، مع الدالة `autoClose`، بغض النظر عن نتيجة الاختبارات، وستغلق الموارد إغلاقًا صحيحًا.

## 10. خلاصة الفصل

يركّز هذا الفصل على كيفيَّة استخدام كوتلن لكتابة اختبارات أنظف وسهلة القراءة، لقد رأينا كيف يعمل إطار KotlinTest الرائع وكيف يمكن توسيعه ليناسب حالات الاستخدام الخاصة بك. يتحسن KotlinTest باستمرار، لذلك تحقق دومًا من صفحة readme على موقع GitHub للاطلاع على الميزات الجديدة التي أُضيفَت إليه منذ تاريخ نشره.

الفصل الثاني عشر:

# الخدمات المصغَّرة مع كوتلن



لا تقتصر كوتلن على تطوير تطبيقات آندرويد فقط، فهنالك الكثير من الشيفرات المكتوبة للواجهة (back-end)، وجميعها مكتوبة بجافا، ولا شيء يمنعك من إضافة كوتلن إلى إليها كلما احتجت إلى إضافة وظيفة جديدة. لا تجعل جافا خيارك الوحيد عندما يتعلَّق الأمر بلغة JVM المراد استخدامها في مشروعك الجديد. عندما يحصل نظامك الموجّه للخدمات المصغّرة (microservices-oriented system) على الضوء الأخضر لبدء البرمجة، فلماذا لا تعتمد على كوتلن؟

ليس الهدف من هذا الفصل هو الغوص عميقًا في عالم تصميم الخدمات المصغَّرة، ولكن الإطلاع على بعض المفاهيم فيه. هنالك الكثير من التوثيقات المكتوبة حول موضع الخدمات المصغَّرة (microservices) وربما قد تعرف بالفعل مبادئها، ومع ذلك، أنا أشجعك على قراءة كتاب Reactive Microservices Architecture: Design Principles for Distributed Systems للكاتب Jonas Bonér، يمكنك الحصول على الكتاب مجانًا بصيغة pdf من دار النشر O'Reilly وذلك عبر هذا الرابط.

سنتعلَّم في هذا الفصل:

- ما هي هيكلية الخدمات المصغَّرة
- لماذا سنستخدم هذا النهج وما هي عيوبه
- إنشاء مشروع Lagom maven للسماح بالبرمجة بكوتلن
- تعريف خدمات Lagom
- تشغيل عنقود تطوير Lagom

## 1. التعريف

تغيَّرت الطريقة التي نصمم البرمجيات ونطورها بها في السنوات القليلة الماضية، فبدأنا الآن بتطوير برامج مصمَّمة لتعمل في السحابة (cloud) تلك الفكرة الوليدة من التطور السريع والحاجة الملحة؛ إذا كنت بالفعل على دراية بهذا العالم، فسوف تتعرَّف على بعض تلك الحاجات التي أسهمت بولادة هذا النهج الذي نتخذه هذه الأيام:

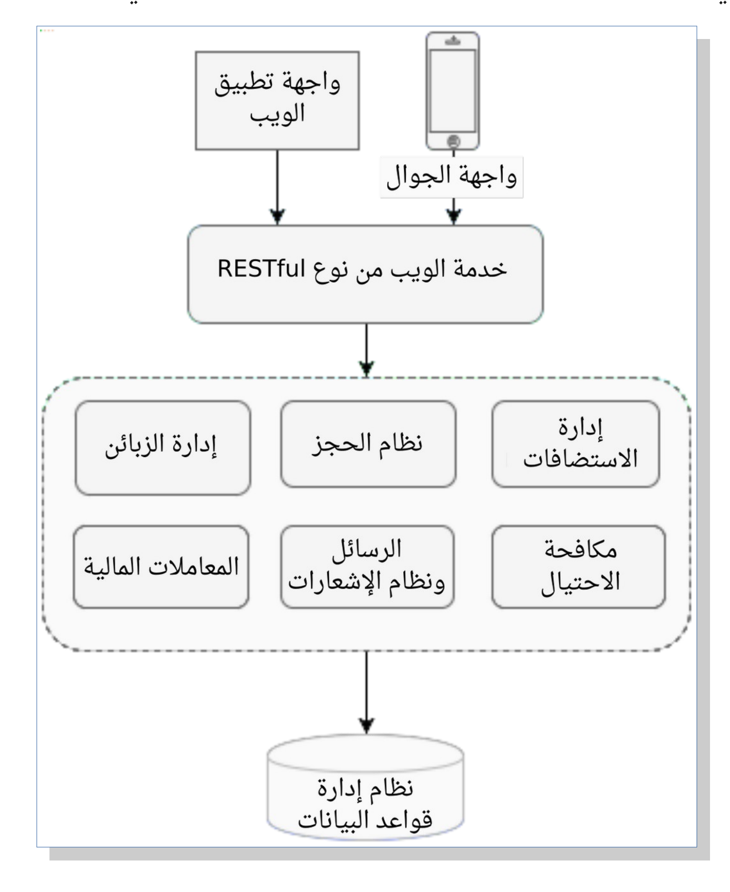
- هنالك طلب لتخفيض التكاليف مع الاستمرار في تحسين الأداء.
- يجب عليك تقديم وظائف جديدة لتلبية المتطلبات التي يفرضها الواقع المتطوِّر المتقلِّب، ويجب أن



يكون التحوُّل سريعًا.

- يجب على البرنـامج الوصـول إلى العملاء حـول العـالم والتعامـل مـع ارتفـاع الطلب الـذي قـد ينتج أثنـاء التشغيل على هذا النطاق.

إن أردت معرفة ما هي الخدمات المصـغَّرة وكيـف وصـلنا إلى مثـل هـذه المبـادئ، لنقم بتمـرين. تخيَّل أن لـديك فكـرة مذهلـة مثـل فكـرة Airbnb، ولم يكن هنالـك شـيء مطبَّق مثلها في السـوق. أنت الآن في وضـع يسـمح لـك بتحديد عرض عالي المستوى لبنية نظامك. عند اتخاذ نهج بسيط للغاية، قد ينتج شيء مماثلًا لهذا:




أنا متأكد من أنك حدَّدت التصميم النموذجي ذي المستويات الثلاثة (three-tier-level design):

- طبقة واجهة المستخدم
- طبقة النطاق/الأعمال
- طبقة التخزين

في البدايـة، يمكن أن يثبت بسـهولة أنـه الحـل المناسـب، إذ يمكنـك تحـزيم التطـبيق ونشـره كتطـبيق واحـد، ويمكنـك حـتى تشـغيل نسـخ متعـدِّدة منه بمجـرَّد ضبط مـوازن حمـل (load balancer) . كـل هـذا يبـدو سـهلًا وصحيحًا. في البداية، لكن تظهر فكرتك الثوريـة وينمـو عملـك بـاطراد (إذا كنت فضـوليًا، اقـترح عليـك إجـراء بحث سريع على نمو قاعدة عملاء Airbnb ورؤية الأرقام المثيرة للإعجاب)، وستفهم أنَّه إذا حقّقت هـذا النجـاح، فيجب عليك دفع المزيد والمزيد من الوظائف والمزايا الجديدة للتعامل مع الطلب المتزايد.

وعندما تفعل ذلك، ستزداد أسطر شيفرتك البرمجيَّة بالإضافة إلى تعقيد برنامجك، وستبدأ بـدفع التكلفـة، مثـل إصلاح المشاكل وإضافة الوظائف وكل هذا سيبدو تحـديًا أكـبر. سـيؤدي ذلـك إلى تقييـد سـرعة التطـوير، ولن تنبهـر بذلك بما أنَّك تمتلك العمل وتشرف عليه مباشرةً، وبالطبع لن تصل أرباحك إلى أقصى إمكانياتها، وسيأخذ النشر وقتًـا أطول نظرًا لأن تطبيقك سيحتاج على الأرجح إلى وقت أطول حتى يعمل بالشكل المطلـوب، ومن ثَمَّ سـتتأثر عمليـة التسليم المستمر (continuous delivery) بشكل كبير. فكِّر أيضًا في الموثوقية (reliability) فكـل الوحـدات الخاصة بك تعمـل تحت نفس العمليَّة. في حين أنَّ مسـألة الموثوقيـة بالتأكيـد من أولى الأولويـات لحـدٍ مـا، فكـل مـا يتطلَّبه الأمر هو حدوث تسريب في الذاكرة (memory leak) في إحدى وحدات النظام لينهار نظامك وبرنامجك بالكامـل. لا أحتـاج إلى إخبـارك كم هـو مـزعج أن تـرى أن الخدمـة غـير متـوفِّرة إذا كنت بحاجـة إليهـا. واحـدة من السلبيات الكبيرة الأخرى من هذا النهج هو إعادة بناء الشيفرات البرمجيَّة. فعملية ترقيـة أو حـتى اسـتبدال إطـار مـا عمليةً مهمةً وحرجةً أيضًا. على الأرجح سيؤثر هذا على التطبيق كله وسيتطلّب اختباره كاملًا كذلك.

هذا هو المكـان الـذي يسـاعد فيـه النهج الجديـد للخـدمات المصـغَّرة. الفكـرة بسـيطة جـدًا وسـيقول البعض أنَّ الخـــدمات المصـــغّرة هي مجـــرَّد كلمـــة طنانـــة لمـــا هـــو معـــروف بـــالمعماريَّة الموجَّهـــة للخـــدمات (Service Oriented Architecture) وتختصـر إلى SOA). اسـتوحيت بنيـة SOA من هـذه المبـادئ الأربعـة المفروضة على المستأجرين (tenants):

- الحدود صريحة

- الخدمات مستقلة
- تتشارك الخدمات المخطط (schema) والعقد وليس الصنف
- تعتمد توافقية الخدمة على سياسة الخدمة (policy)

ستتعرَّف في الفقرتين التاليتين على مبادئ الخدمات المصغَّرة وستلاحظ الفرق الحقيقي في بمعمارية SOA يكمن في حجم ونطاق الخدمات.

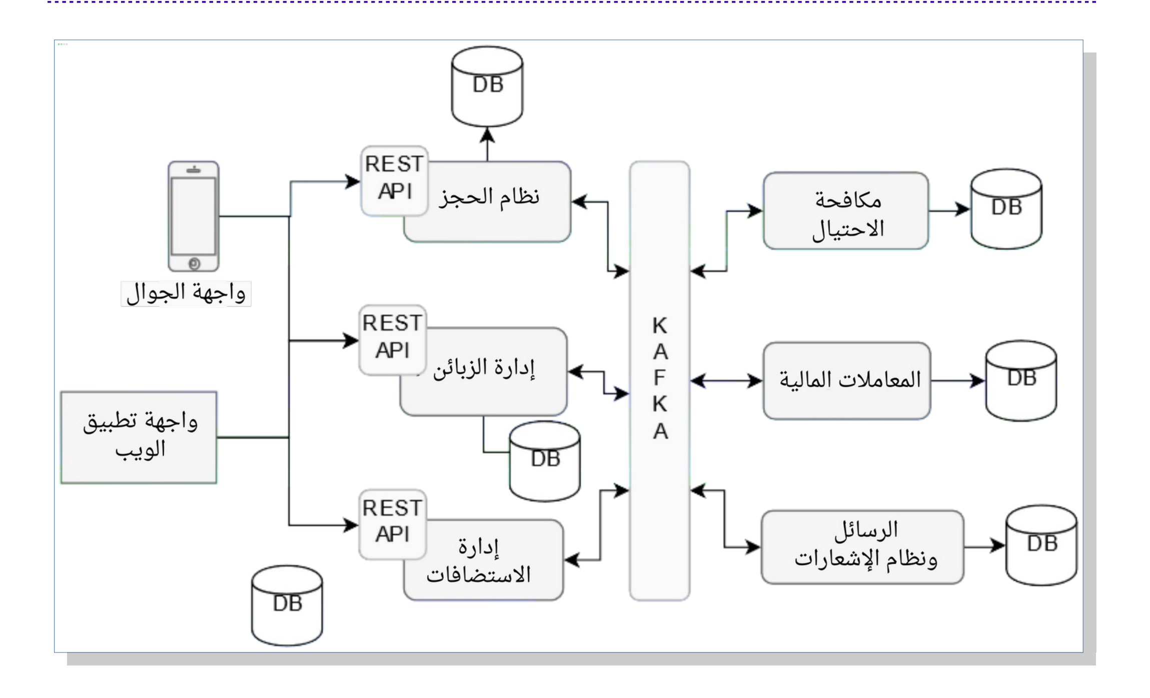
لتجنُّب وعثاء الطريق، من المفترض أن تقسّم تطبيقك الخاص إلى عدة خدمات صغيرة، تركّز كل واحدة منها على وظيفة معيَّنة، وتتفاعل تلك الخدمات مع بعضها بعضًا من أجل توفير جميع متطلبات عمل التطبيق.

لا يوجد تعريف قياسي للخدمات المُصغَّرة، فعند البحث في الإنترنت ستجد تعريفات مختلفة حول ماهيتها وما تفعله، لكن جميعها تشترك في بعض الأشياء، إذ كل خدمة:

- يمكن تطويرها باللغة والإطار الذي يعجبك.
- تتواصل الخدمات المُصغَّرة فيما بينها باستخدام بروتوكول واضح المعالم وعلى مجموعة من الواجهات.
- توفِّر سيناريو عمل واحد فقط.
- يمكن توفير إصدار لكل خدمة بشكل مستقل عن الأخرى.
- يمكن نشرها وترقيتها بشكل مستقل.
- يمكن توسيعها.
- يمكن أن تحمل أي خدمة حالة (state)، إذا لزم الأمر.
- تتجنب حالات الفشل.
- تُبلغ عن الحالة الحاليَّة والمقاييس وعمليات التشخيص.

سيعطيك كل ما سبق ذكره فكرة عن الخدمات المصغَّرة.

غطينا الآن تعريف الخدمات المصغَّرة، فدعنا نرجع إلى الرسم البياني عالي المستوى للنظام الشبيه بخدمة Airbnb؛ ما يلي هو نهج واحد لتجزيء تطبيق متآلف إلى أجزاء أصغر:




ربما لاحظت أن مخزن قاعدة البيانات في المخطط السابق قد أزيلت، وهناك سبب لذلك، وهو أنَّ ووجود قاعدة بيانات لكل خدمة هو الحل الأفضل لضمان الاقتران بين الخدمات المصغَّرة،، بهذه الطريقة لا يمكنك الحصول على نموذج بيانات (data model)على مستوى التطبيق وبهذا سينتهي بك المطاف ببيانات متكرِّرة بالتأكيد، لكن هنالك بعض الأسباب التي تدفعك لفعل شيء مثل هذا، فهي نتيجة لعدم اقترانها بإحكام بالمكونات ( components). بهذه الطريقة، يمكن لكل خدمة العمل مع نسخة قاعدة البيانات وليس بالضرورة أن تكون كل قواعد البيانات من نفس النوع. يمكنك الحصول على عميل وخدمات استضافة تعمل مع قاعدة بيانات مثل MongoDB أو RethinkDB في حين أن خدمة الدفع تعتمد على قاعدة بيانات MySQL، فالحصول على مخطط بيانات مختلف لكل قاعدة بيانات لا يؤثر على الخدمات الأخرى. وسيضيف هذا المزيد من التعقيد في عمليات التطوير (DevOps، اختصار إلى Development Operations) عن طريق الاضطرار إلى صيانة أكثر من قاعدة بيانات.

ستتحدَّث العديد من المراجع حول معمارية الخدمات المصغَّرة عن وجود «استدعاء إجراء بعيد» (remote

procedure call ويختصر إلى RPC) كوسيلة للتواصل بين الخدمات، ويهدف هذا الاستدعاء توفير التواصل بين العمليات، إذ تستطيع التطبيقات عبرها استدعاء دوال عن بعد ويمكنها أيضًا تنفيذ عمليات على الحاسوب نفسه المتواجدة فيه أو على حواسيب موجودة على الشبكة نفسها. النهج النموذجي هو إمكانية استدعاء كل مكوِّن حيث يملك نقط نهايات REST مكشوفة من أجل إطلاق (trigger) مجموعة من الإجراءات. ستنتهي بك السُبل مع زيادة تعقيد برنامجك بأنابيب بيانات (data pipelines) متشابكة كالسباغيتي؛ إذا كنت تعرف إطار Kafka فستعرف التحديات التي واجهها موقع LinkedIn والأسباب التي بني من أجلها. إذا لم يكن هذا شيئًا مألوفًا لديك، فأرجو منك أن تطلع عليه أو تستمع إلى بعض محادثات Jay Kreps.

## 2. العيوب والمساوئ

لا توجد تكنولوجيا مثاليّة ولدى الخدمات المصغّرة بعض السلبيات، كما يقول العمل فريد (Fred Brooks)، لا توجد رصاصة فضية (No Silver Bullet).

لقد ذكرنا أن الخدمات المصغَّرة لديها مخطط قاعدة بيانات خاص بها، لذلك لا يمكنك الحصول على تحديثات انتقالية (transactional updates) على نطاق وحداتك كما تفعل عادةً عند التعامل مع تطبيق أحادي النوى ( monolithic application) يملك قاعدة بيانات واحدة. سينتهي الأمر في نهاية المطاف إلى أن يكون نظامك متناسقًا وذلك لا يكون إلا بمواجهة تحديات كبيرة.

إن تنفيذ تغيير يمس أكثر من خدمة واحدة له تعقيده الخاص. في التطبيقات أحادية النواة، ستكون هذه الأشياء سهلة للغاية، كل ما عليك القيام به هو تغيير الوحدات المطلوبة ومن ثم نشر كل منها دفعة واحدة، لكن في نظام الخدمات المصغَّرة، توجد تبعيات مشاركة بين الخدمات يجب ترقيتها. سينتهي بك الأمر إلى نشر الخدمة التي تعتمد عليها جميع الخدمات الأخرى أولًا ثم تكرر هذه الخطوة حتى تترقى آخر خدمة مطلوبة. لحسن الحظ، هذه التغييرات ليست شائعة جدًا، عادة ما تكون التغييرات قائمة بذاتها إلى خدمة واحدة فقط.

بما أنَّنا نتحدَّث عن النشر، دعنا نتوسّع في سلبيات نظام مع معمارية الخدمات المصغّرة، فإن نشر تطبيق أحادي النواة هو بسيط للغاية لأنه ينطوي على توزيع قطع البناء، وإذا كان التوافر العالي (high availability) مطلوب، فستفعل ذلك على مجموعة من الخوادم تملك موازن حمل في واجهتها، تختلف الأمور كثيرًا بالنسبة لنظام يتكوَّن من العديد من الخدمات المنفصلة. تختلف الأمور كثيرًا بالنسبة لنظام يتكون من خدمات منفصلة عديدة، بعض

التطبيقات المعروفة لديها المئات من الخدمات، وكل واحدة من هذه الخدمات تملك نسخ عديدة لضمان تـوافر دائم. إن عمليّة التكوين، أو النشر، أو التوسيع أو المراقبة تتطلَّب المزيد من التحكم في النشر ومستوى عـالي من التشـغيل التلقـائي، ولتحقيـق هـذا، ربمـا قـد تسـتعمل حـل PaaS، أو تنفيـذ حـل خـاص بـك من خلال المـزج بين Docker و Kubernetes.

أحد التحديات الأخرى التي تواجه معماريَّة الخدمات المصغّرة تأتي عند محاولة اختبـار تطبيقـك. فعنـد اختبـار التطبيق أحادي النواة سيكون الأمر سهلًا للغاية لكن بالنسبة إلى الخدمات المصغّرة ستكون الأمور مختلفة قليلًا. في هذه الحال، تحتاج الاختبارات إلى إطلاق الخدمة وربـط جميـع الخـدمات التابعـة لهـا، وبطبيعـة الحـال، يتطلّب هـذا المزيد بذل المطور جهدًا كبيرًا.

هذه النقاط يجب أن لا تجعلك تستسلم وتتجنَّب اعتماد معماريّة الخدمات المصـغَّرة، فالفوائـد تفـوق السـلبيات بكثير، وإذا كانت الشركات الكبيرة تستعملها، فيجب أن يكون هنالك شيء أو اثنين يدفها على الأقل لفعـل ذلـك. على جانب آخر، لم تستخدم أوبر Uber معمارية الخدمات المصغَّرة، لكنهـا تنتقـل ببطء إلى هـذا الأسـلوب راغمـةً بسـبب حاجة العمل الملحة. لذلك، خطط جيّدًا في البداية للتأكد من أن المعمارية تعمل على نمـو نشـاطك التجـاري بـدلًا من وقوفها عائقًا في وجهه.

## 3. لماذا الخدمات المصغَّرة؟

ربما تشاءمت برؤيتك لقائمة العيوب التي قدَّمناها آنفًا، وحان الوقت لتعرف أنَّ هنالك فوائد جمةً يدفعك لتبني هذا النهج.

واحدة من الفوائد الرئيسيَّة لتصميم لمثل هذا التصميم هو كسر تعقيد التطبيق أحـادي النـواة، وسـينتهي الأمـر بتوفير مجموعة محدودة من الخدمات التي تسمح بمجملها بتحقيق نفس الوظيفة مع وجـود شـيفرة برمجيَّة سـهلة الفهم والصيانة والتطوير.

ستجد أنَّه مع نهج الخدمات المصغَّرة، فإنك لا تقتصر على تكنولوجيا أو لغة محدَّدة لتنفيـذ كامـل التطـبيق بهـا، إذ يمكن تطوير كل خدمة على حدة باستقلالية كاملة عن الأخرى وعبر فريق واحد منفصل، ويمكن للفريق أن يختـار أكثر التقنيـات ملائمـة لتنفيـذها بهـا. فكم من مـرَّة أردت اسـتخدام إطـار/لغـة أحـدث لأنَّهـا ستضـيف قيمـة جديـدة لتطبيقك كاملًا لكنك عالق بإطار أو لغة قديمة يكلف تغييرها مبلغًا كبيرًا لا داعي له آنذاك مثلًا؟

يدعم تصميم الخدمات المصغّرة النشر المستمر (continuous deployment)، فيمكن نشر الخدمات المصغَّرة بشكل مستقل، بمجرَّد اختبار التغييرات، يمكن نقلها إلى بيئة الإنتاج مباشرة. ونظرًا لأنَّ النشر معزول ومنفصل، يمكنك توفير عدّة ترقيات لتطبيقك خلال اليوم نفسه دون إيقافه.

لتحقيق الإنتاجيَّة، يمكن قياس كل خدمة مصغَّرة على حدة، وعلاوة على ذلك، يمكن ضبط خادم استضافة لخدمتك على أساس الموارد المطلوبة فقط، فإذا كانت الخدمة تتطلَّب الكثير من الذاكرة، فيمكنك الحصول على خادم بذاكرة أكبر دون زيادة الموارد الأخرى مثل المعالج والعكس بالعكس يقاس، وبالتالي تخفيض التكاليف الخاصة بك.

هنالك أسباب أخرى تجعلك تصمّم نظامك باستخدام مبادئ الخدمات المصغَّرة، لكن يجب عليك أن تذهب وتقرأ بحوثًا متعمّقة حول الموضوع، على الأقل، الآن، يجب أن تكون القيمة المضافة من خلال النهج قد اتضحت.

## 4. إطار العمل Lagom

يُعد Lagom إطار JVM جديد من Lightbend لكتابة الخدمات المصغَّرة؛ هذا إطار جديد ومفتوح المصدر صدر في بداية عام 2016، وفي وقت مراجعة وتدقيق هذا الكتاب، أُصدِرت النسخة 1.6.1، ويمكنك العثور على الشيفرة المصدرية على github.com/lagom/lagom ومن هنالك يمكنك الانتقال إلى موقع الإطار والذي يحتوي على المزيد من التفاصيل والمعلومات.

يأتي Lagom مع دعم لأربع مميزات رئيسيَّة هي: الواجهة Service البرمجية، والواجهة Persistence البرمجية وبيئة تطوير وبيئة إنتاج.

يجري التصريح عن الخدمات وتنفيذها عبر الواجهة Service ليستخدمها العميل؛ يسمح مكوِّن تحديد موقع الخدمة باكتشاف خدمتك، وعلاوة على ذلك، تسمح لك الواجهة باستعمال بروتوكول طلب-استجابة متزامن بالإضافة إلى مجرى تدفُّق غير متزامن.

توفّر الواجهة `Persistence` دعمًا لاستمرار كيانات النطاق (domain entities) في خدماتك. يعتني Lagom بتوزيع تلك الكيانات المستمرَّة عبر مجموعة من العقد (nodes)، مما يمكن المشاركة والتوسيع الأفقي مع قاعدة بيانات Cassandra؛ لا شيء يمنعك من توصيل أي نوع تخزين تريده. مصطلح «المشاركة» (

sharding)، بالنسبة للذين لا يعرفونه، هي طريقة لتقسيم البيانات لقاعدة بيانات على عدَّة أجهزة، والسبب في ذلك هو توزيع الحمل وتحقيق توسيع خطي (linear scaling) من أجل تلبية متطلبات الأداء.

تسمح لك بيئة التطوير بتشغيل جميع خدماتك ودعم Lagom للبنية التحتيَّة من خلال استخدام أمر واحد. ويمكن للمطور باستخدام Lagom إحضار خدمة جديدة أو الانضمام إلى فريق تطوير Lagom في دقائق معدودة.

يوفِّر Lagom دعمًا جيدًا لبيئة الإنتاج عن طريق Lightbend ConductR والذي يسمح بنشر خدمات Lagom ومراقبتها وتوسيع نطاقها في البيئة الحاوية.

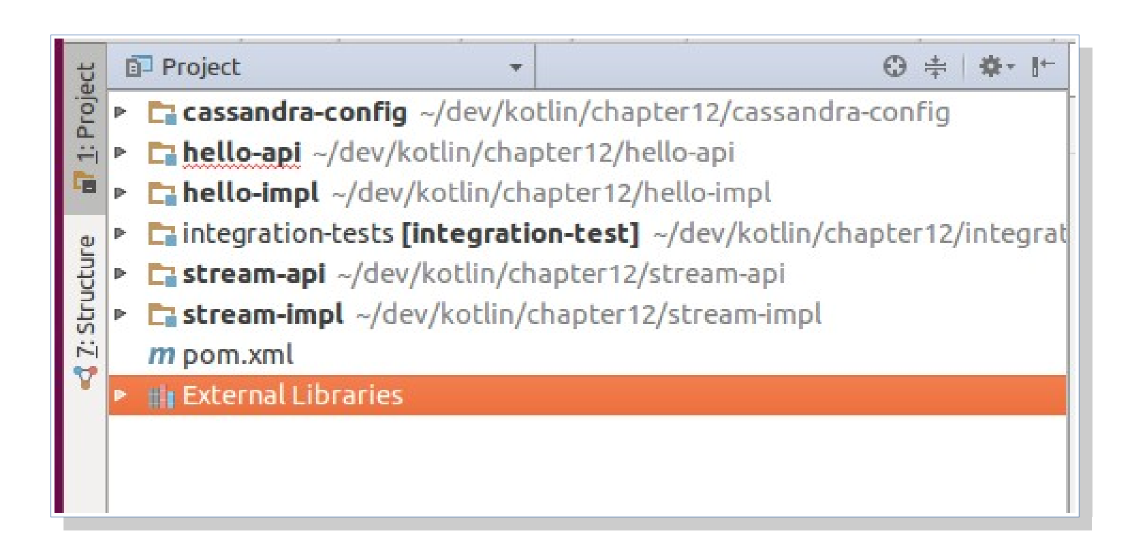
يدعم Lagom بشكل كامل لأداتين من أدوات البناء المتوفّرة في السوق: SBT و Maven. بما أنَّ SBT ليس شائعًا كما Maven، فسنركِّز على الأخيرة. أسهل طريقة للبدء وتعلم Lagom هو الاستفادة من الإضافة Maven archetype لتوليد مشروع أساسي. بمجرَّد أن تعتاد على التخطيط والتبعيات (dependencies)، فستنشئ ملف pom بنفسك. بما أنَّه ليس هنالك دعم لمشروع يعتمد على كوتلن وقت كتابة هذا الكتاب[14]، فسنضطر إلى إلى البدء مع مشروع جافا ومن ثم تعديل pom لتمكين كوتلن. من المتوقع أنَّ يكون Apache Maven مثبتًا على جهازك. إن لم يكن، يرجى اتباع التعليمات على موقع Apache Maven لتثبيته.

من نافذة الطرفيَّة، اكتب الأمر التالي:

```
mvn archetype:generate -DarchetypeGroupId=com.lightbend.lagom \
-DarchetypeArtifactId=maven-archetype-lagom-java -DarchetypeVersion=1.2.0
```

سيطالبك بتوفير قيمة للحقول `GroupId` (ليكن `com.programming.kotlin`) و `ArtifactId` (ليكن `chapter12`) و `version` (اتركه فارغًا)، بمجرَّد تشغيل هذا، يجب أن تشاهد هيكل تخطيط المجلَّد هذا:

14 قد يضحي الدعم متوافرًا في وقت قراءتك لهذا الكتاب، لذا تحقق منها.




نحتاج بعد ذلك إلى إضافة ملحق Maven لمصرِّف كوتلن، لقد غطينا هذا بالفعل في الفصل الأول، لذلك دعنا نذهب ونعدِّل ملف pom وفقًا لذلك.

بالنسبة لهذا المشروع، سنستخدم Kotlin 1.1-M04، وهذه آخر نسخة للإصدار 1.1 وقت كتابة هذا الكتاب، إذا حاولت استخدامه مع أي من النسخ 1.0.x فلن يعمل لأن دعم جافا 8 يأتي فقط مع الإصدار 1.1. قد تعرف الآن أن هذا الإصدار يأتي مع مجموعة من التحسينات، والأهم من ذلك، دعم الروتين المشترك.

| تنبيه | حدثنا إصدار كوتلن في بداية الكتاب إلى الإصدار 1.3.31 ولكن لم نُحدِّث هذا الفصل منذ كتابة الكتاب على عكس الفصول السابقة. على أي حال، انتبه إلى إصدار كوتلن وجافا وحاول استعمال أحدث إصدار مع تحديث الشيفرة وفقًا لذلك، وحتى إن جرى تحديث الإصدارات إلى أحدث ما يمكن، فستختلف في الوقت الذي ستقرأ الكتاب فيه، إذ التقنية سريعة التطور. |
|---|---|

يفتقد مطوّر #c هذا النوع من الوظائف عندما تبرمج لآلة جافا الافتراضية JVM. أول شيء يجب عليك فعله هو إضافة خاصيَّة جديدة في ملف pom الخاص بك لإصدار كوتلن:

```
<properties>
    ...
    <kotlin.version>1.1-M04</kotlin.version>
    <lagom.version>1.2.0</lagom.version>
    ...
</properties>
```

لا تُخزّن الإصدارات البارزة في مستودع Maven، وسنحتاج إلى إضافتها له من أجل الحصول على التبعيات. ستحتاج إلى إضافة التالي إلى ملف pom:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.programming.kotlin</groupId>
    <artifactId>chapter12</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>pom</packaging>

    <pluginRepositories>
        <pluginRepository>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
            <id>bintray-kotlin-kotlin-dev</id>
            <name>bintray</name>
            <url>http://dl.bintray.com/kotlin/kotlin-dev</url>
        </pluginRepository>
    </pluginRepositories>

    <repositories>
        <repository>
            <snapshots>
```

```
            <enabled>true</enabled>
          </snapshots>
          <id>bintray-kotlin-kotlin-dev</id>
          <name>bintray</name>
          <url>http://dl.bintray.com/kotlin/kotlin-dev</url>
        </repository>
      </repositories>
...
```

بعد ذلك، نحتاج إلى تمكين مصرّف كوتلن والسماح بتشغيل مصرّف جافا بعده أثناء بناء Maven، لذلك غيّرنا التالي في ملف pom. أرجو منك مراجعة الفصل الأول للحصول على المزيد من التفاصيل حول سبب الحاجة إلى ذلك.

```
<build>
<plugins>
...
<plugin>
  <groupId>com.lightbend.lagom</groupId>
  <artifactId>lagom-maven-plugin</artifactId>
  <version>${lagom.version}</version>
 </plugin>
 <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.5.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
    <compilerArgs>
      <arg>-parameters</arg>
    </compilerArgs>
```

```
    </configuration>
    <executions>
        ...
    </executions>
</plugin>
</plugins>
...
```

لا تقلق بشأن كتابة هذا بنفسك، ستجد كل شيء في الشيفرة البرمجيّة المرافقة للكتاب. نأمل قريبًا أننا سنرى قالب Maven archetype يسمح بمشروع Lagom بدعم كوتلن[15].

حتى الآن، يمكننا بناء الشفرة المصدريّة وتشغيل الخدمات، من الطرفيّة، شغّل السطر التالي لبدء جميع الخدمات:

```
mvn lagom:runAll
```

سيصرف هذا الشيفرة البرمجيّة، يشغّل الاختبارات، ويبدأ جميع الخدمات، وسنغطي كل واحدة منها الآن. للتحقق من أن كل واحدة منها على ما يرام، اكتب `localhost:9000/api/hello/World` في متصفحك ويجرب أن ترى Hello, World مطبوعةعلى الشاشة. هنالك نقطتي نهاية لـ /api/hello كما ستعرف لاحقًا. للتحقق من أن HTTP POST يعمل، أوقف عمل أمر `curl`:

```
curl -H "Content-Type: application/json" -X POST -d '{"message":"Hi"}'
http://localhost:9000/api/hello/Gabriela
```

بما أننا فعّلنا كوتلن، فسنترجم الشيفرة البرمجية من جافا إلى كوتلن، لنبدأ بمشروع `hello-api` وصنف `GreetingMessage`، ستحتاج في البداية إلى إنشاء مجلد kotlin، أحد أخوة مجلد Java، والذي سيحتوي على الملفات المصدرية لكوتلن، وبعد ذلك أضف الحزمة التالية إلى المجلد الجديد: `com.programming.kotlin.chapter12.hello.api`. أنشئ ملف كوتلن جديد وألصق الشيفرة البرمجية التالية، وعلّق (ضع تعليقات) لملف جافا المطابق:

15 يمكن أن يكون هنالك دعم متوافر لذلك أثناء قراءتك لهذا الكتاب.

```
@JsonDeserialize data class GreetingMessage(val message: String)
```

تعد الأصناف الثابتة مهمة لـ Lagom وهنالك عدد من الأماكن التي تحتاج إلى استخدامها:

- أنواع طلبيات Service واستجاباتها.
- أوامر كيانات Persistent وأحداثها وحالاتها.
- رسائل النشر والاشتراك (Publish and subscribe messages).

إذا كنت تستخدم جافا وتريد معالجة الشيفرة المتداولة فستحتاج إلى استخدام وإعـداد أداة Immutables (immutables.github.io) لكن لحسن الحظ، تسمح لنا كوتلن بإنشاء صنف غير قابل للتغيير بسرعة مع التوابع equals، و hashCode و toString المولَّدة لنا. تذكر، إذا كان الصنف غير القابل للتغيير يحتوي على عضو وهـو تجميعة، فيجب على هذه التجميعة أن تكون غير قابلة للتغيير إذا كان الصنف يوفِّر جالب getter لها وخلاف ذلـك فإن ضمان عدم قابليَة التغيير سَخرَق، وكما تعلم بالفعل، تقدم كوتلن دعمًا للتجميعات غير قابلة للتغيير في المكتبـة القياسيَّة، لذلك من السهل جدًا تقديم صنف غير قابل للتغيير حتى إذا كان يحتوي على تجميعة.

لنترجم HelloService.java بعد ذلك إلى كوتلن:

```
interface HelloService : Service {
   fun hello(id: String): ServiceCall<NotUsed, String>

   fun useGreeting(id: String): ServiceCall<GreetingMessage, Done>

   override fun descriptor(): Descriptor {
      val helloCall: (String) -> ServiceCall<NotUsed, String> = {
this.hello(it) }
      val helloGreetings: (String) -> ServiceCall<GreetingMessage, Done> =
{ this.useGreeting(it) }

       return named("hello")
       .withCalls(
         pathCall("/api/hello/:id", helloCall),
```

```
        pathCall("/api/hello/:id", helloGreetings))
       .withAutoAcl(true)
   }
}
```

لنركز الآن على مشروع `hello-impl` ونوفِّر تنفيذ كوتلن للصنف `HelloServiceImpl` (تحتاج إلى تعليق تنفيذ جافا المولّد لك):

```
class HelloServiceImpl
@Inject
constructor(private val persistentEntityRegistry:
PersistentEntityRegistry) : HelloService {
   init {
      persistentEntityRegistry.register(HelloEntity::class.java)
   }

   override fun hello(id: String): ServiceCall<NotUsed, String> {
      return ServiceCall<NotUsed, String>{
         val ref = persistentEntityRegistry.refFor(HelloEntity::class.java,
id)
         ref.ask<String, Hello>(Hello(id, Optional.empty<String>()))
      }
   }

   override fun useGreeting(id: String): ServiceCall<GreetingMessage,
Done> {
      return ServiceCall<GreetingMessage, Done>{
         val ref = persistentEntityRegistry.refFor(HelloEntity::class.java,
id)
         ref.ask<Done, UseGreetingMessage>(UseGreetingMessage(it.message))
      }
   }
}
```

إذا شغّلت `mvn lagom:runAll` في الطرفيَّة، فستجد أن كل شيفرة تغيَّرت قد حُدِّدت لك، فالملحق Lagom Maven هو المسؤول على مشاهدة تغييرات الملف وإعادة تصريف التعليمات البرمجيّة تلقائيًا. مع التغييرات السابقة، سيظهر لك خطأ الآن أثناء محاولة بدء تشغيل الخدمات مرَّة أخرى:

```
Error in custom provider, java.lang.IllegalArgumentException:
Service.descriptor must be implemented as a default method
   at
com.lightbend.lagom.javadsl.server.ServiceGuiceSupport.bindServices(Servic
eGuiceSupport.java:33) ...
Caused by: java.lang.IllegalArgumentException: Service.descriptor must be
implemented as a default method at
com.lightbend.lagom.internal.api.ServiceReader$ServiceInvocationHandler.in
voke(ServiceReader.scala:280)
```

تتحقق الشيفرة المصدريَّة حيث رُمي الاستثناء أولًا من أنَّ التابع `descriptor` يحتوي على التابع الافتراضي لواجهة جافا 8، وإذا لم يكن الأمر كذلك، تتحقق مما إذا كان قد أُنشئ مع سكالا (Lamom مكتوب بلغة Scala و على إطار Akka). لسوء الحظ، فإن Java 8 interop قادم إلى كوتلن وهو ليس كاملًا الآن ولا يعالج توابع الواجهة الافتراضيّة. لا توجد طريقة لإعلام المصرّف بتابع `descriptor` في الشيفرة البرمجيَّة السابقة الذي يجب أن يصرّفها كتابع جافا 8 الافتراضي. ولذلك في الوقت الحالي، يجب علينا أن نعتمد على جافا لتعريف واجهة الخدمة، وبالتالي، إذا ألغيت تعليق شيفرة جافا لـ `HelloService` وحذفت تنفيذات كوتلن، ستكون قادر على تشغيل البيئة مرَّة أخرى[16].

بعد ذلك سنترجم مشروع `api-impl` إلى كوتلن، أغلب الشيفرة البرمجيَّة سهلة التغيير، فسينتهي الأمر بتقليل عدد الأسطر قليلًا، وسترى أدناه الملفات الجديدة لـ `HelloCommand` و `HelloEvent` و `HelloState`:

```
//HelloCommand.kt
      interface HelloCommand : Jsonable
```

16 وجب التنبيه إلى أنَّ المعلومات الواردة في هذه الفقرة ذات صلاحية محدودة، وقد تكون انتهت وجاءت إصدارات لاحقة لما ذُكِر وتوافر دعم أثناء قراءتك لهذه النسخة المترجمة من الكتاب لم يكن متوافرًا أثناء كتابة الكتاب وترجمته.

```
    @JsonDeserialize data class UseGreetingMessage @JsonCreator
constructor(val message: String) : HelloCommand, CompressedJsonable,
PersistentEntity.ReplyType<Done>
    @JsonDeserialize data class Hello @JsonCreator constructor(val name:
String, val organization: Optional<String>) : HelloCommand,
PersistentEntity.ReplyType<String>

    //HelloEvent.kt
    interface HelloEvent : Jsonable

    @JsonDeserialize data class GreetingMessageChanged @JsonCreator
constructor(val message: String) : HelloEvent

    //HelloState.kt
    @JsonDeserialize data class GreetingMessage @JsonCreator
constructor(val message: String)
```

يعتبر الصنف `HelloEntity` الأكثر صعوبة للتحويل بشكل طفيف، فلا يفعل المحرِّر وظيفة كبيرة عند تحويل شيفرة جافا إلى كوتلن، وينتهي الأمر بإنتاج شيفرة برمجيّة لا تُصرَّف. لأنه من الصعب تحويله من جافا (المحرِّر مضلل يشير إلى الموضع الخطأ من الشيفرة معتقدًا أنها من سببت الخطأ)، سندرجه بعد قليل، في وقت كتابة هذه السطور، لم يصدر Kotlin 1.1 ولم يكن IDE متوافقًا بشكل كامل مع الإصدار القادم، أنا متأكد في الوقت التي تقرأ فيه هذه السطور، سينتهي هذا كله.

```
class HelloEntity : PersistentEntity<HelloCommand, HelloEvent,
HelloState>() {
    override fun initialBehavior(snapshotState: Optional<HelloState>):
PersistentEntity<HelloCommand, HelloEvent, HelloState>.Behavior {

    val b = newBehaviorBuilder(snapshotState.orElse(HelloState("Hello",
LocalDateTime.now().toString())))

    b.setCommandHandler(
```

```
            UseGreetingMessage::class.java,
            { cmd, ctx ->
            ctx.thenPersist(
               GreetingMessageChanged(cmd.message),
               { evt -> ctx.reply(Done.getInstance()) })
            }
         )

         b.setEventHandler(GreetingMessageChanged::class.java,
            { evt -> HelloState(evt.message, LocalDateTime.now().toString()) }
         )

         b.setReadOnlyCommandHandler(Hello::class.java,
            { cmd, ctx -> ctx.reply(state().message + ", " + cmd.name + "!") }
         )
         return b.build()
      }
   }
```

يجب أن تكون قادرًا على التعامل مع بقيـة التحـويلات بنفسـك، يمكنـك دائمـا التحقـق مـع الشـيفرة المصـدرية المتوفرة لهذا الفصل.

قبل الانتقال، شغّل أمـر `curl` المـذكور سـابقًا في هـذا الفصـل ليقـوم بـ `post` على `/api/hello`، سـتلاحظ رمي استثناء لأنه لا يمكن تعـيين لحمولـة `JSON` لصـنف GreetingMessage. يحـدث هـذا حـتى على الـرقم من إضافة التوصيفات json إلى الصنف، نحن بحاجة إلى التعامل مع التسلسل وإلغاء التسلسل بأنفسنا.

أولًا، نحتـاج إلى إضـافة تبعيَّة وحـدة مكتبـة Jackson Kotlin (يقـدم واحـدًا من أفضـل الـدعم لـ JSON لـ JVM) إلى مشروع hello-api. نحن نستخدم الإصدار 2.7.8 لأن إصدار Lagom هذا يستخدم مكتبـة Jackson الأساسيّة:

```
<dependency>
   <groupId>com.fasterxml.jackson.module</groupId>
   <artifactId>jackson-module-kotlin</artifactId>
   <version>2.7.8</version>
   <exclusions>
      <exclusion>
         <groupId>org.jetbrains.kotlin</groupId>
         <artifactId>kotlin-stdlib</artifactId>
      </exclusion>
      <exclusion>
         <groupId>org.jetbrains.kotlin</groupId>
         <artifactId>kotlin-reflect</artifactId>
      </exclusion>
   </exclusions>
</dependency>
```

الخطوة التالية هي إنشاء نسخة `ObjectMapper` عبر نمط المفرد:

```
object Jackson{
   val mapper: ObjectMapper = {
      ObjectMapper().registerKotlinModule()
   }()
}
```

لـدعم تسلسـل صـنف `GreetingMessage`، يجب توفـير صـنف وراثـة `StrictMessageSerializer`، الشيفرة البرمجيَّة واضحة للغاية كما يمكنك أن ترى هنا:

```
class GreetingMessageSerializer : StrictMessageSerializer<GreetingMessage>
{
     internal var serializer:
MessageSerializer.NegotiatedSerializer<GreetingMessage, ByteString> =
object : MessageSerializer.NegotiatedSerializer<GreetingMessage,
ByteString> {
     override fun protocol(): MessageProtocol {
```

```
        return MessageProtocol().withContentType("application/json")
      }

      @Throws(SerializationException::class)
      override fun serialize(messageEntity: GreetingMessage): ByteString {
        return
ByteString.fromArray(Jackson.mapper.writeValueAsBytes(messageEntity))
      }
   }

   internal var deserializer =
MessageSerializer.NegotiatedDeserializer<GreetingMessage, ByteString> {
      bytes -> Jackson.mapper.readValue(bytes.iterator().asInputStream(),
GreetingMessage::class.java)
   }

   override fun serializerForRequest():
MessageSerializer.NegotiatedSerializer<GreetingMessage, ByteString> =
serializer
   @Throws(UnsupportedMediaType::class)
   override fun deserializer(protocol: MessageProtocol):
MessageSerializer.NegotiatedDeserializer<GreetingMessage, ByteString> =
deserializer

   @Throws(NotAcceptable::class)
   override fun serializerForResponse(acceptedMessageProtocols:
List<MessageProtocol>):
MessageSerializer.NegotiatedSerializer<GreetingMessage, ByteString> =
serializer
}
```

للسماح للإطار باستخدام هذا الصنف، يجب أن تكون شيفرة المصدريّة لـ HelloService descriptor أن تعدّل لتعيين map الصنف للتسلسل-إلغاء التسلسل. يوفر توثيق Lagom عرضًا تفصيليًا لكيفية تقديم تسلسل مخصّص، إذا أردت معرفة المزيد، يرجى قراءة التوثيق.

```
@Override
     default Descriptor descriptor() {
        return named("hello").withCalls(
           namedCall("hi", this::sayHi),
           pathCall("/api/hello/:id", this::hello),
           pathCall("/api/hello/:id", this::useGreeting)
        ).withAutoAcl(true)
        .withMessageSerializer(GreetingMessage.class, new
GreetingMessageSerializer());
    }
```

الآن نحن نحظى بدعم كامل لمشروع قائم على Lagom باستخدام كوتلن، يمكنك إعادة إطلاق الخـدمات عن طريق أمر `maven` والتحقق من أنها تعمل.

## 5. تعريف الخدمات

عند كتابة مشروعك، يجب عليك الالتزام بأفضل الممارسات والمعايير، ويجب وضع واجهة أي خدمة (service interface) في مشروع جديد باسم `api-*`، فيمكنك رؤية الواجهة `HelloService` تتبع نفس القاعدة، الشرط هو أن تمدّد واجهة الخدمة الخاصة بك واجهة الخدمة Lagom وتوفِّر تنفيذًا افتراضيًا للتابع `descriptor`، إذ سيكون `descriptor` هو المسؤول عن تعيين الخدمة إلى بروتوكول النقل الأساسي.

```
public interface HelloService extends Service {
   ServiceCall<NotUsed, String> hello(String id);
   ServiceCall<GreetingMessage, Done> useGreeting(String id);

   @Override
   default Descriptor descriptor() {
      return named("hello").withCalls(
         pathCall("/api/hello/:id",  this::hello),
         pathCall("/api/hello/:id", this::useGreeting)
      ).withAutoAcl(true);
```

```
    }
}
```

هذا هو وصفٌ لخدمة تعرض استدعاء خدمتين: الأولى `hello` والثانية `useGreeting`، يرجع كلا التابعين نسخة من `ServiceCall` تمثِّل مقبضًا إلى استدعاء التابع الذي يمكن استدعاؤه عند استهلاك الخدمة. إليك تعريف الواجهة `ServiceCall`:

```
interface ServiceCall<Request, Response> {
     CompletionStage<Response> invoke(Request request);
}
```

تعريف الواجهة هذه بسيط، وكما ترى، فإنه يغلِّف مفهوم نموذج طلب-الرد (request-response).

يجب أن يكون تنفيذ ملتزمًا ببروتوكول النقل المُستخدَم، وهو البروتوكول HTTP كما في الشيفرة البرمجيّة المولّدة، ومع ذلك، لا شيء يمنعك من تغيير ذلك إلى `WebSockets` أو أي بروتوكول آخر يناسب احتياجاتك بشكل أفضل.

يجب أن يحمل كل استدعاء لخدمة معرِّفًا، إذ يربط هذا المعرِّف الاستدعاء بالتابع المناسب في الواجهة. يمكن أن تتخذ هذه المعرفات شكل أسماء ثابتة أو مسارات (كما رأينا في التعليمات البرمجيَّة المولَّدة)، لكن يمكن حمل المعاملات الحيويَّة المسلَّمة إلى توابع استدعاء الخدمة.

المعرف البسيط هو المعرِّف عبر التابع `namedCall`، وكما يوحي الاسم، ستسمي استدعاء الخدمة الخاصة بك. دعنا نوسّع `HelloService` لتضمين مثل هذا الاستدعاء؛ أُضيف التابع `sayHi` إلى الواجهة، وتستخدم الشيفرة هذه المرة `namedCall` بدلًا من `pathCall`:

```
public interface HelloService extends Service {
   ServiceCall<NotUsed, String> sayHi();
   ...
   @Override
   default Descriptor descriptor() {
```

```
      return named("hello").withCalls(
        namedCall("hi", this::sayHi),
        ...
      ).withAutoAcl(true);
    }
}
```

يجب الآن تحديث الصنف `HelloServiceImpl` لتوفير تنفيذ للتابع `sayHi`:

```
override fun sayHi(): ServiceCall<NotUsed, String> = ServiceCall<NotUsed,
String>{
  completedFuture("Hi!")
}
```

بمجرَّد إعادة التصريف، يجب أن تتمكن من لصق عنوان URL التالي: `http://localhost:9000/hi` إلى المتصفح ورؤية الكلمة !Hi على الشاشة.

يستخدم المشروع النموذجي هذا المعرِّفات المستندة إلى المسار (path-based identifiers)، إذ يمكنك رؤية ذلك في مقتطف الشيفرة البرمجيّة `HelloService` السابق. يستخدم هذا النوع من المعرفات مسار URI واستعلام سلسلة نصيّة لتوجيه الاستدعاءات، ويمكن اختياريًا توفير المعاملات التي يمكن استخراجها، وإذا عملت مع خدمة REST، ستكون على دراية بالمفهوم، فإذا افترضنا أنك تعمل على نظام تأمين صحي وتريد أن يكون العميل قادرًا على إعادة بوليصة التأمين، فقد يبدو تعريف الخدمة الخاص بك على النحو التالي:

```
ServiceCall<NotUsed, PSequence<Customer>> getDependents(long
policyHolderId, int pageNo, int pageSize);

default Descriptor descriptor() {
  return named("customers").withCalls(
    pathCall("/customer/:policyHolderId/dependencies?pageNo&pageSize",
this::getDependents)
  );
}
```

سيضمن لك إطار Lagom ربط تابع الخدمة بأنواع توابع REST مناسبة؛ اكتب http://localhost:9000 في متصفحك وستحصل على قائمة بجميع نقاط النهاية المتاحة:

```
GET \Q/hi\E Service: hello ( http://0.0.0.0:57797 )
GET \Q/api/hello/\E([^/]+) Service: hello ( http://0.0.0.0:57797 )
POST \Q/api/hello/\E([^/]+) Service: hello ( http://0.0.0.0:57797 )
GET \Q/stream\E Service: stream ( http://0.0.0.0:58445 )
```

يمكنك أن تلاحظ أنَّ كلا من POST و GET مدعومين لمسار api/hello/، وقد تتساءل أين عُرِّف ذلك؟ يتوفر هذا بواسطة إطار Lagom نفسه، سننظر إلى تعريف ServiceCall وإذا كان تعيين المعامل الوارد ليس NotUsed، فسيعيِّن استدعاء الخدمة إلى نوع طلب POST.

يوجد نوع أخير من مُعرِّف الاستدعاء يدعمه الإطار، وهو مُعرِّفات استدعاءات REST وهي تقدم لك تعريفًا أكثر دقّة لربط خدمة الاستدعاء الخاصة بك باستدعاءات REST، فتخيَّل أنَّ لديك خدمةً لتوفير عمليات CRUD وهي Create، و Read، و Update وDelete لقاعدة العملاء، فستأخذ واجهة الخدمة الخاصة بك الشكل التالي:

```
ServiceCall<Customer, NotUsed> addCustomer(long customerId);
ServiceCall<NotUsed, Customer> getCustomer(long customerId);
ServiceCall<NotUsed, NotUsed> deleteCustomer(long customerId);
default Descriptor descriptor() {
   return named("orders")
   .withCalls(
      restCall(Method.POST,   "/api/customer", this::addCustomer),
      restCall(Method.GET,    "/api/customer/:customerId/",
this::getCustomer),
      restCall(Method.PUT,    "/api/customer/:customerId",
this::updateCustomer)
      restCall(Method.DELETE, "/api/customer/:customerId",
this::deleteCustomer)
   );
}
```

ربما قد لاحظت في تعريف descriptor وجود الصنف NotUsed، ويرشد هذا النوع الإطار أنَّ المعامل

الوارد أو كائن الاستجابة الصادرة لن يُستخدَم. إذا نظرت إلى تعريف حذف أحد العملاء، ستلاحظ أن كليهما لا يستخدمان الطلب والاستجابة، جميع أمثلة التعليمات البرمجيَّة المقدمة حتى الآن تستخدم ما يعرف باسم الرسائل الصارمة (strict messages). فتُعدُّ رسالةٌ ما صارمةً إذا كان من الممكن ربطها بصنف JVM؛ هنالك نوع آخر من الرسائل، وهو مجرى التدفق stream، بنيت وظيفة التدفُّق على واجهة Akka Streams. وستتعرَّف على رسالة التدفُّق عندما ترى استخدام النوع `Source`. توفِّر الواجهة البرمجية طريقةً سهلةً لوصف إعدادات معالجة بيانات التدفق، ويمكن تنفيذها بكفاءة وبموارد محدودة الاستخدام.

```
ServiceCall<String, Source<String, ?>> gbpToUsd()
ServiceCall<Source<String, ?>, Source<String, ?>> chatRoom()
```

يصف الإدخال الأول مجرًى أحادي الاتجاه (unidirectional stream) في حين أنَّ الثاني ثنائي الاتجاه (bidirectional stream). بشكل افتراضي، إنَّ Lagom يستخدم `WebSockets` لتوفير طبقة النقل لمجرى البيانات لكن لك الحريَّة في ربط آلية النقل الخاصة بك. لن نتعمَّق في تفاصيل دعم Lagom للمجاري، لكن يمكنك الذهاب ومعرفة المزيد عبر الرجوع إلى توثيق الإطار.

## 6. تنفيذ خدمة Lagom

لقد رأينا بالفعل كيفيَّة تعريف الخدمة، الخطوة الطبيعية التالية هي تنفيذ واجهة الخدمة، وهذا ما يتحقق في مشروع `hello-impl`. إنَّ الصنف `HelloServiceImpl` مسؤولٌ عن تنفيذ الواجهة البرمجية API الخاصة بك، لقد رأينا هذا عبر مقتطفات الشيفرة البرمجيَّة المذكورة سابقًا. ستلاحظ شيء مهم وهو عدم تنفيذ جميع التوابع الاستدعاء فعليًا، بل ترجع مقبض تابع (method handle) عبر نسخة `ServiceCall`، السبب وراء هذا النهج هو السماح بتركيب دالة (function composition) من أجل تحقيق خط أنابيب معالجة (processing pipeline) حيث يمكن إضافة المصادقة (authentication) والتخويل (authorization) والتسجيل (logging) ومعالجة الاستثناءات (exception handling) بسهولة.

لننقل التركيز مرَّة أخرى إلى الصنف `ServiceCall`، من التعريف المقدَّم بالفعل، يمكنك أن ترى أنه يأخذ كائن الطلب ويرجع `CompletionStage<Response>`، هذه الحاوية ليست سوى وعد يقوم به API، وفي مرحلة ما في المستقبل،ستُحسَب الاستجابة وستكون متاحة للاستهلاك، ويمكنك تغيير نوع الاستجابة عبر تابعي الواجهة

البرمجية thenApply و thenCompose. وسينتج هذا بالطبع وعدًا آخر، النوع CompletionStage وتوابعها تسمح لك ببناء تطبيقات تفاعلية غير متزامنة بالكامل. ينبغي أن يكون تنفيذ تابع sayHi المقدَّم في وقت سابق أكثر منطقيَّة الآن.

بمجرَّد تنفيذ الخدمة الخاصة بك، ستحتاج إلى تسجيلها مع إطار العمل من أجل الاستفادة منها. بُني Lagom على إطار Play والذي يسمح لك ببناء تطبيقات الويب القابلة للتطوير باستخدام جافا أو سكالا، وافتراضيًا، يستخدم الإطار Guice كإطار حقن التبعيّة وبالتالي يعتمد Lagom عليه أيضًا. وهذا هو سبب وجود صنف HelloModule ويرث من AbstractModule (صنف خاص بـ Guice) و ServiceGuiceSupport (صنف خاص بـ Lagom).

```
class HelloModule : AbstractModule(), ServiceGuiceSupport {
   override fun configure() {
      bindServices(serviceBinding(HelloService::class.java,
HelloServiceImpl::class.java))
   }
}
```

يمكن أن يأخذ تابع bindServices أمثلة ServiceBinding متعدّدة، ومع ذلك، قدمنا في المثال واحد فقط، ولكن لا ينبغي أن يمنعك هذا من توفير أكبر عدد ممكن من أربطة الخدمات (service bindings) حسب حاجتك، ولكن تأكد من استدعاء bindServices مرّة واحدة، وإلا سينتهي بك الأمر بخطأ Guice runtime-configuration.

عندما تبدأ العمل مع Lagom، ستتعرَّف على مصطلحات Event Sourcing (ES) و Command Query Responsibility Segregation (CQRS).قبل تحديد ما هو ES، دعنا نرى السمات الأساسيَّة للحدث:

- يمثّل نشاطًا تجاريًا، فكر في حجز رحلة طيران، ستقول "حُجز مقعد على متن الرحلة BA0193 من قبل Alex Smith".
- تحمل بعض معلومات الوصف معها، تحمل معلومات المثال السابق، وتضيف بيانات إلى الحدث الخاص بك في شكل معلومات شخصيَّة، بدل الأمتعة، مواعيد الرحلة، وما

إلى ذلك.

- هي رسائل غير قابلة للتغيير وذات اتجاه واحد، فالناشر، والذي هو في هذه الحالة موقع الحجز، سيبث الرسالة و N من المشتركين سيتحصلون عليها.
- حدث في الماضي، عند وصف حدثًا، ستستخدم فعل الماضي دائمًا.

يعدُّ ES نهج لاستمرار حالة التطبيق من خلال تخزين التاريخ الذي يحدِّد حالته الحالية، ففي مثال شراء تذكرة الطيران، سيتتبع نظام الحجز عدد الحجوزات المكتملة للرحلات الجويَّة المعنيَّة والمقاعد المتبقيَّة المتاحة؛ هنالك خيارات لتتبع المقاعد المتاحة: إما البدء مع المجموع المتاح وتقليل هذا العدد حتى يصل إلى صفر، أو جمع الحجوزات الحاليَّة دائمًا لمعرفة ما إذا كانت قد وصلت إلى الحد الأقصى لعدد المقاعد المتاحة على الطائرة.

قد تسأل نفسك ما فائدة ES، بخلاف إنشاء سجل تدقيق لجميع معلوماتك، إذا كنت تطوِّر برنامجًا لمؤسسة ماليَّة، فهذا بالفعل مكسب مهم، ومع ذلك، هنالك فوائد أخرى أيضًا:

- الأداء: بما أن الأحداث غير قابلة للتغيير، فيمكن الاستفادة في الكتابة من وضع الإلحاق فقط (append-only)، مما يجعلها أسرع في التخزين، ولكن يجب الانتباه عن طريق استخدام هذا النهج لأنَّك تحتاج إلى إعادة إنشاء حالة التطبيق عن طريق المرور من خلال جميع السجلات، وقد يكون هذا مكلفًا وغير مقبول أحيانًا، ومع ذلك، هنالك طرق للتغلب على السلبيات.
- البساطة: يمكن أن يوفر عليك تخزين الأحداث تعقيد التعامل مع نماذج المجال المعقدَّة.
- القدرة على إعادة قراءة البيانات: لديك تسلسل الأحداث والعلل، دعنا نقول أن بعض القيمة المصرَّفة خطأ، يمكن حلها عن طريق المرور من خلال البيانات المُخزَّنة وتطبيق قاعدة برمجيَّة جديدة.

بالطبع يأتي ES مع تحدياته الخاصة، فتحتاج إلى التأكد من أن النظام الخاص بك يمكنه التعامل مع إصدارات متعدّدة من نوع رسالة/حدث، فكيف يمكنك التعامل مع استعلامات معقَّدة، تخيَّل نظامًا مثل Airbnb، سؤال واحد قد ترغب في الإجابة عليه هو: ما هي جميع الحجوزات التي تمت لروما في يوليو 2015 بسعر ليلة أعلى من 100 يورو؟ هذا هو المكان الذي سيتدخَّل فيه CQRS.

CQRS هو نمط الذي نشئ في نمط Domain Driven Design (DDD) وهذا الأخير هو نهج لتصميم

نظم معقدَّة مع قواعد العمل المتغيِّرة باستمرار، فتحلَّل أنت مشكلة المجال لإنتاج نموذج مفـاهيمي (conceptual model)، والذي يصبح الأساس الخاص بك لحلك.

لن نتعمّق أكثر في DDD (فهو أكبر من نطـاق هـذا الفصـل)، لكننـا سـنحدِّد بعض المصـطلحات المحـدَّدة الـتي ستواجهها أثناء العمل مع Lagom، عند تحديد نموذج المجال الخاص بك، ستستخدم الشروط التالية:

- الكيانات: الكائنات التي يمكن وصفها بواسطة الهوية لن تتغيَّر أبدًا، ومرجع حجز الرحلة هو واحد منها.
- قيمة-كائنات: لا يمكن لجميع الكائنات الخاصة بك أن تكون كيانـات، ولهـذه الكائنـات، قيمـة سـماتها مهمـة، على سبيل المثال، لا يوجد نظام حجز رحلات لتوفير معرِّف فريد لعنوان العميل.
- الخدمات: لا يمكنك تصميم كل شيء ككائن، سيعتمد نظام الدفع على الأرجح على معالج دفع طرف ثالث، وعلى الأرجح سـتبني خدمـة دون حالـة، وهي مسـؤولة عن تمريـر جميـع المعلومـات المطلوبـة من قبـل الطرف الثالث لمعالجة الدفع.

في عالم DDD، يحدد مصطلح «التجميع» (aggregate) كتلة من الكيانـات (entities) وكائنـات القيمـة (value objects) والتي تشكل حدود الاتساق داخل النظام. الكيانات وكائنات القيم مرتبطة ببعضها بعضًا.

للوصول إلى الكائنات داخل التجميع (aggregate) يجب المرور من خلال جـذر التجميـع، وهـذا هـو حـارس البوابة (gatekeeper)، ومن خلال التجميع، يصف DDD ويدير جزء مجموعة علاقات النـوع من نمـوذج النطـاق النموذجي.

وبـالعودة إلى نمـط CQRS، فـإنَّ الهـدف هـو تخصـيص المسـؤولية للتعـديل والاسـتعلام عن أنـواع الكائنـات المختلفة، ينتهي الأمر بإنشاء نموذج الكتابة والقراءة، وبما أنَّ الكائنات لديها مسؤولية واحدة، إمـا لتعـديل أو لقـراءة البيانات، فإنَّها ستجعل شيفرتك أبسط وأسهل.

والآن، بعـــد أن غطينـــا الســـياق، يمكننـــا التحـــدث بمزيـــد من التفاصـــيل حـــول الكيانـــات المســـتمرَّة (persisting entities)، فستحصـل في الشـيفرة المصـدريَّة المولَّدة على الصـنف `HelloEntity` المشـتق من `PersistentEntity`، وهو صنف أساسي يقدِّم معرِّفًا لا يتغيَّر أبـدًا، ومن خلال هـذا المعرفـة، يمكن الوصـول إلى أي كائن، ومن وراء الكواليس، يستخدم الإطار ES لاستمرار الكيان، وبالطبع، تُسجِّل جميع التغييرات التي تطـرأ على الحالة في التسلسل الذي ظهرت به من خلال إلحاقها بسجل الأحداث، فالكيان المستمر هو ما يعادل جذر التجميـع (

aggregate root) الذي ناقشناه آنفًا.

يأتي التفاعل مع PersistentEntity في شكل رسالة أمر (command message)، هذه هي الرسائل التي ترسلها وتعالجها واحدة تلو الأخرى، ويمكن أن تؤدي رسالة الأمر إلى تغيير الحالة، والذي سيُسجِّل بعد ذلك:

```
b.setCommandHandler(UseGreetingMessage::class.java, {
   cmd, ctx -> ctx.thenPersist(GreetingMessageChanged(cmd.message), {
      evt -> ctx.reply(Done.getInstance()) })
})
```

يجب عليك توفير معالج أمر لكل نوع من الرسائل، في هذه الحالة، لدينا واحد فقط، كل معالج أوامر مسؤول عن إرجاع نسخة Persist، والذي يصف الحدث (الأحداث) الذي سيستمر إذا كان هنالك مثل هذا الشرط، نموذج التعليمات البرمجيّة هي تعليمات استمرار لحدث واحد فقط، ومع ذلك، هنالك دعم استمرار أكثر من حدث واحد من خلال thenPersistAll.

إذا لم تكن الرسالة الواردة صحيحة، فيمكنك رفضها باستخدام ctx.invalidCommand أو ctx.failedCommand، فعلى سبيل المثال، إذا كانت GreetingMessage تحتوي على حقل رسالة فارغ فسترغب في رفض الأمر.

لا تعدِّل جميع الرسائل الحالة (state)، فهذه مُهمّة رسائل الاستعلام (query messages)، وفي هذه الحالة، سنستخدم setReadonlyCommandHandler لتسجيل معالج الأوامر الخاص بك:

```
b.setReadOnlyCommandHandler(Hello::class.java,{
   cmd, ctx -> ctx.reply(state().message + ", " + cmd.name + "!") })
```

بمجرَّد تخزين الحدث، يُحدَّث مخزن الكيان الحالي من خلال الجمع بين الحالة الحاليّة والحدث الوارد؛ ستستخدم setEventHandler لتسجيل المعالج الخاص بك، والذي يجب أن يعود بحالة جديدة غير قابلة للتغيير، ويوفِّر PersistentEntity تابع حالة للحصول على حالة الكيان الأخيرة. يجب عليك توفير معالج أحداث حالة لكل نوع رسالة أمر، في الشيفرة البرمجيّة السابقة، هنالك نوع رسالة واحد فقط وكل ما يفعله هو الاحتفاظ بالرسالة الأخيرة ووقت الاستلام:

```
b.setEventHandler(GreetingMessageChanged::class.java, {
   evt -> HelloState(evt.message, LocalDateTime.now().toString())
})
```

عادةً، ستشغل بعض نسخ خدمتك وسيُحدَّد موقع كل كيان تلقائيًا إلى واحدة من العقد، إذا كانت العقدة تنخفض، فإنَّ تلك الكيانات المخصَّصة لها سيعاد توجيهها إلى عقدة عمل أخرى، ومن الجيِّد أنَّ أمر `producer` لا يجب أن يعرف الموقع الحالي. سيهتم الإطار بإعادة التوجيه إلى العقدة المناسبة، ولأغراض التحسين، سيخزِّن الإطار الكيان وحالته إذا أُستخدِم، وسيحرِّر الموارد بعد عدم الوصول إليها في وقت معيَّن. عند تحميل كيان، فإنَّه قد يعيد تشغيل جميع الأحداث المخزَّنة أو سيستخدم نهج اللقطة للحصول على أحدث حالتها.

## 7. خلاصة الفصل

تعلمت في هذا الفصل كيف أن بنية الخدمة المصغَّرة تبني نظامًا كمجموعة من الخدمات الصغيرة والمعزولة عن بعضها، إذ تملك كل خدمة بياناتها الخاصة ويمكن تغيير حجمها بشكل مستقل لتوفير مرونة للفشل. تتفاعل هذه الخدمات مع بعضها بعضًا لتشكيل نظام متماسك. كان هذا الفصل بمثابة مقدّمة سريعة لإطار العمل Lagom، والذي هو إطار جديد لتطوير الخدمات المصغَّرة التفاعلية على JVM، هنالك الكثير من الأشياء التي يمكننا التحدث عنها عندما يتعلَّق الأمر بـ Lagom، فهو يحتاج إلى كتاب بأكمله، ونأمل، بهذه المقدمة السريعة أنَّنا أشعلنا فيك الفضول لتذهب وتتعلم المزيد عن هذا الإطار، فلقد تعلمت كيف تفعِّل كوتلن لمشروع Lagom، ويمكنك الآن الاستمرار والاستفادة من جميع فوائد كوتلن لزيادة سرعة تطوير نظام التوزيع الخاص بك القادم.

سيكون الفصل الأخير من هذا الكتاب هو مقدِّمة للتزامن، سنطّلع على المصطلحات وكيف يمكن حل مشاكل التزامن الشائعة في كوتلن، وستقرأ عن Akka وستكتشف ما هو وكيف يساعد على التزامن وبالطبع كيف يمكن دمجه مع كوتلن.

الفصل الثالث عشر:

# التزامن



على الأرجح سـمعت بقـانون مـور (Moore's Law)، ففي عـام 1965، لاحـظ جـوردن مـور أن عـدد الترانزستورات التي يمكن وضعها في دارة كهربائية (شريحة معالج) لكل بوصة مربعة قد تضاعفت كل عام مرة منذ اختراعها، لذا أعطي اسم قانون مور على الاعتقاد بأنَّ هذا الأمر سيستمر، وفي الحقيقة كـان هـذا صـحيحًا ويتحقـق تقريبًا كل 18 شهرًا، وحتى الآن. نتيجة لذلك فـإن أجهـزة الحاسـوب أصبحت أسـرع وأصـغر وتسـتخدم طاقـة أقـل، ودليل ذلك وجود الهواتف المحمولة في كل مكان.

ومع ذلك، لا شيء يدوم إلى الأبد، النمو الأسي (exponential growth) في سياق قـوَّة المعالجـة يـتراجع بالفعل، فإذا كنا غير قادرين باستمرار على جعل الأنظمة تعمل بشكل أسرع عن طريق زيـادة السـرعة الصـافية (raw speed)، فيجب البحث عن بديل.

أحـد هـذه البـدائل هـو تقسـيم الـبرامج إلى أجـزاء يمكن تشـغيلها بشـكل مـتزامن (concurrently) ومن ثم استخدام معالجات متعـدّدة. معًا، يمكن لمجموعـة من شـرائح المعالجـة بطيئـة الأداء أن تعمـل أسـرع ممـا لـو كـانت شريحة المعالجة واحدة وذلك ما دامت البرامج قادرة على تنفيذ الشيفرات على التوازي فيمـا بينهـا للاسـتفادة منهـا جميعا. يشار إلى مجموعة الشرائح المتوضعة على وحدة المعالجة المركزيَّة (CPU) على أنَّها معالج متعـدِّد النـواة (multicore processor9).

تتطلَّب هيكلة البرامج التي يُسمَح بتشغيلها في وقت واحد مناهج وتقنيات جديدة، ونجد أنَّ العديد من الأفكار المستندة إلى مفهوم التزامن ليست جديدة وقـد كـانت موجـودة منـذ السـبعينات، والجديـد هـو أنَّ اللغـات الحديثـة تسمح لنا باستخدام هذه الأفكار بسهولة أكبر مما نستطيع في اللغات منخفضة المستوى في غابر العصور.

التزامن (Concurrency) هو موضوع كبير لا يسعه قسم واحد ويجب تخصيص كتاب كامل لـه، لـذلك يركـز هذا الفصل على بعض الأساسيات الأساسية فقط، فسيغطي هذا الفصل المواضيع التاليَة:

- مفهوم الخيوط (Threading)
- عملية المزامنة (Synchronization) والمراقبين (monitors)
- الأنواع الأساسية المتزامنة (Concurrency primitives)
- التقنيات غير متزامنة (Asynchronous techniques) والتقنيات الغير معطلة أو الحاجزة (non-blocking techniques)

## 1. الخيوط

الخيط (thread، أو النَيْسب[17]) هو واحد من أبسط اللبنات الأساسيَّة للشيفرة المتزامنة، فهو جزء من البرنامج الذي يمكنه تنفيذ خيوط متزامنة بالنسبة إلى بقيَّة الشيفرة البرمجيَّة. يمكن لكل خيط مشاركة الموارد كالذاكرة ومقابض الملفات، وفي النظام الذي يسمح بإنشاء الخيوط، تنقسم كل عمليَّة على خيط واحد أو أكثر، وإذا لم يستفيد البرنامج من الخيوط لتنفيذ شيفرته تزامنيًا، فسيسمى بعملية الخيط الواحد أي يعد عمليةً واحدةً ويُنفَّذ عبر خيط واحد.

في نظام وحدة المعالجة المركزيَّة الواحد (single CPU system)، تتداخل الخيوط المتعدَّدة في وقت المعالجة الذي يُستقطَع لها (time slicing)، مع تلقي كل خيط مقدار وقت قصير من وقت المعالج يسمى مقدار المعالجة (quantum أو مقدار معالجة الخيط Thread Quantum)، إذ تحدث عملية استقطاع هذا المقدار من الوقت بسرعة كبيرة غير ملحوظة، ويبدو كما لو أن الخيوط تعمل في نفس الوقت، فعلى سبيل المثال، قد يحدِّث خيط واحد الملف في حين يعيد آخر رسم النافذة على الشاشة. بالنسبة للمستخدم، تظهر الخيوط وكأنَّها تعمل بالتوازي ولكنها في الحقيقة تعمل بالتسلسل كما شرحنا آنفًا. ويطبَّق نفس المبدأ على تشغيل العمليات باستخدام مُجدول نظام التشغيل (operating system scheduler).

عند انتهاء وقت المعالجة المُستقطَع لخيط، يُنتقَل إلى الخيط التالي، وعندما يكتمل تنفيذ خيط، يجلب مُجدوِل الخيوط (thread scheduler) خيطًا آخر لتنفيذه بدلًا منه، ويسمى هذا بسياق تبديل الخيط (thread-context-switch) وهو مشابه لمبدل السياق (context switch) الذي تمر به العمليات (processes). إن مبدِّل سياق الخيط أخف من مبدل عملية كاملة، هذا لأنَّ الخيوط تتشارك في موارد كثيرة وبالتالي تكون البيانات المراد معالجتها مُخزَّنة وجاهزة.

تلميح: التزامن (Concurrency) هو مصطلح عام يعني أن مهمتين نشطتين تتقدمان في نفس الوقت، في حين أن التوازي (parallelism) هو مصطلح أكثر صرامة، لأنَّ ذلك يعني أن المهمتين تنفذان في لحظة معينة معًا.

---

17 النَيْسبُ هو تسمية تُطلَق على مسرى من النمل السائر أي الطريق الذي يُرَى من خلالها عند مسيرها، وهي الترجمة الأدق برايي -يقول المُحرِّر- لأنَّها تطابق تمامًا العملية أو التسمية التي تصفها. على أي حال، آثرت استعمال «خيط» وهي الترجمة الشائعة في موسوعة حسوب، إلى حين اعتماد هذه الترجمة وشيوعها.

عـادةً مـا يُسـتخدَم أحـدهما عوضًـا عن الآخـر والعكس لكن التـوازي الحقيقي هـو هـدف البرمجـة التزامنيـة (concurrent programming).

في JVM، يرتبط كل كائن Thread بحالة (state)، فيمتلك الخيط حالة واحد في وقت محـدَّد، وتُسـرد هـذه الحالات في الجدول التالي:

| الحالة | الوصف |
|---|---|
| `NEW` | أنشئ الخيط لكن لم يبدأ استخدامه بعد. |
| `RUNNABLE` | الخيط في هذه الحالة يعمل من وجهة نظر JVM، وهذا لا يعني بالضرورة أنَّه ينفذ أي شيء من الشيفرة، فقد يكون في انتظار مورد من نظام التشغيل، ويقتطع آنذاك وقتًا من المعالجة دون فعل شيء. |
| `BLOCKED` | يسمى الخيط الذي ينتظر الحصول على ملكيّة مورّد بمراقب (monitor). |
| `WAITING` | الخيط دخل في حالة الانتظار، ولن يستيقظ في هذه الحالة حتى تخطره بعض الخيوط الأخرى بذلك. |
| `TIMED_WAITING` | يشبه هذا حالة WAITING باستثناء أن الخيط هنا سيخرج من حالة الانتظار بعد مرور فترة من الوقت، إذا لم يُخطِره أحد. |
| `TERMINATED` | الخيط قد خرج وانتهى. |

## أ. الحجب والتعطيل

يستهلك الخيـط الـذي يعمـل مـوارد وحـدة المعالجـة المركزيَّة (CPU)، وإذا لم يتمكن خيـطٌ قيـد التشـغيل من إجراء تقدم، فيعني أنَّه يستمسك بموارد يمكن أن تُخصَّص لخيط آخر يريد أن يستفيد منها هو الآخر. مثال على ذلك

خيط يقرأ بيانات من الشبكة، فالقراءة من الشبكة اللاسلكية يمكن أن يكون أبطأ بـ 1000 إلى 10000 ضعف من القراءة من الذاكرة RAM، ولذلك يقضي الخيط غالبية الوقت بانتظار استلام البيانات من الشبكة.

في تنفيذ الخيوط الغبي، فإنَّ الخيط سيعصى داخل حلقة تكرار متحققًا من وجود المزيد من البايتات حتى اكتمال العملية أو التحقق مما إذا كان الخيط قد أنهى وقت المعالجة المُستقطَع له، وهذا مثال عن انشغال الخيط، فعلى الرغم من وجود خيط مشغول من الناحية الفنيّة (مستخدمًا وقت وحدة المعالجة المركزيّة)، فإنَّه لا يفعل أي شيء مفيد.

في JVM، يمكننا أن نشير إلى أنَّ الخيط غير قادر حاليًا على التقدم ومن ثم سحبه من مجموعة الخيوط المؤهلة للجدولة، ويسمى هذا بتعطيل الخيط أو حجبه (blocked). الميزة الآن هي حينما يُحجَب الخيط، يتخطاه مجدول الخيوط وبذلك لن يضيع وقت المعالج هباءً منثورًا.

تقوم العديد من عمليات الإدخال/الإخراج في المكتبة القياسيّة بعملية الحجب، مثل `InputStream.read()` أو `Thread.sleep(time)` أو `ServerSocket.accept()`.

## إنشاء خيط

تمتلك كوتلن دالة تابعة لجعل عملية إنشاء خيط سهلة للغاية، تُسمَى هذه الدالة ذات المرتبة الأعلى (top-level function)، وهي جزء من مكتبة كوتلن القياسية، باسم `thread` ببساطة، وتقبل دالة مُجرَّدة للتنفيذها بالإضافة إلى العديد من المعاملات المسماة للتحكم في إعداد الخيط:

```
thread(start = true, name = "mythread") {
   while (true) {
      println("Hello, I am running on a thread")
   }
}
```

في المثال السابق، أنشأنا خيطًا مسمًى (named thread) يبدأ في التنفيذ على الفور، وإذا أردنا تأخير تنفيذ الخيط حتى بعض الوقت في المستقبل، فيمكننا تخزين مقبض (handle) في نسخة `thread` ومن ثم استدعاء `start`:

```
val t = thread(start = false, name = "mythread") {
   while (true) {
      println("Hello, I am running on a thread sometime later")
   }
}
t.start()
```

إذا لم تسمي الخيط، فسيحصل على الاسم الافتراضي الذي يوفره له JVM.

## إيقاف خيط

سيتوقف الخيط بشكل طبيعي بمجرَّد إرجاع الدالة المُجرَّدة المُمرَّرة له، ولوقف الخيط بشكل وقائي أو إجباري، فلا ينبغي لنا استخدام الدالة `stop` الموجودة في الصنف `Thread`، فقد توقف استخدام هذه الدالة منذ سنوات، ويجب علينا بدلًا من ذلك استخدام شرط يمكننا التكرار عليه، أو فاستخدم مقاطعة (interrupt) إذا كان الخيط يستدعى دوال حاجبة (blocking functions) والسماح للدالة المُجرَّدة بالإرجاع لنعود للحالة الأولى.

بالنسبة إلى الحالة السابقة، صرحنا عن خاصية `var` تسمى `running`، والتي ضبطناها إلى القيمة `true`، ومن ثم، نسمح لأي شيفرة تريد إيقاف هذا الخيط بضبها إلى القيمة `false`. نحتاج بذلك إلى تحقق الخيط من قيمة هذا المتغيِّر (الذي يمثِّل حالة)، وإلا فإنَّ الخيط قد يصل إلى الحالة التي لا يتوقف فيها أبدًا:

```
class StoppableTask : Runnable {

   @Volatile var running = true

   override fun run() {
      thread {
         while (running) {
            println("Hello, I am running on a thread until I am stopped")
         }
      }
   }
}
```

**ملاحظة**

من النقاط المهمة التي يجب ذكرها هنا هو استخدام التوصيف `@Volatile` على متغيِّر الحالة، وهذا الأمر بالغ الأهمية لضمان أنَّ حالة المتغيّر تنتشر بين الخيوط. فبدون هذا التوصيف، فقد يحدِّد الخيط الخارجي المتغيِّر إلى القيمة `false`، ومع ذلك، فقد لا يرى الخيط الحاوي لها هذه القيمة أي لا يرى التغيير المجرى عليها، وهذا جزء من نمط ذاكرة جافا (Java Memory Model وتختصر إلى JMM)، والذي هو خارج نطاق هذا الكتاب، ولكن إذا كنت مهتمًا، فإنَّه عند البحث على الإنترنت على JMM فستحصل على ما يكفي من المعلومات لفهمه.

إذا كان لدينا خيط يستدعي نداءات مُعطِّلة (blocking calls)، فإنَّ استخدام المتغيِّر `running` وحده لن يعمل بسبب أنَّ الخيط قد يكون محجوبًا أو مُعطَّلًا عندما تُضبَط قيمة `running` إلى `false`. فكر في هذا المثال للمنتج والمستهلك:

```
class ProducerTask(val queue: BlockingQueue<Int>) {

   @Volatile var running = true
   private val random = Random()

   fun run() {
      while (running) {
         Thread.sleep(1000)
         queue.put(random.nextInt())
      }
   }
}

class ConsumerTask(val queue: BlockingQueue<Int>) {

   @Volatile var running = true

   fun run() {
```

```
      while (running) {
        val element = queue.take()
        println("I am processing element $element")
       }
    }
}
```

يتشارك كل من المُنتِج (producer) والمستهلك (consumer) طابورًا (queue)، هو نسخة BlockingQueue، والذي يوفِّر دوالًا حاجبة للحصول لجلب قيم ووضعها في الطابور، take() و put() على التوالي، فإذا لم تكن هنالك عناصر يمكن جلبها من الطابور، فيُعطَّل الخيط ويُحجَب حتى تتوفر قيمة. لاحظ أنَّ الخيط ينام في حالة المُنتِج، فهو مصمم لإبطاء سرعة المُنتِج، وهذا مثال على مُنتِج بطيء ومستهلك سريع.

لبدء المثال، ننشئ نسخ للمهام ونبدأ مستهلكين بتنفيذ عدة مستهلكات ومنتج واحد، كل واحد في خيط بمفرده:

```
val queue = LinkedBlockingQueue<Int>()

val consumerTasks = (1..6).map { ConsumerTask(queue) }
val producerTask = ProducerTask(queue)

val consumerThreads = consumerTasks.map { thread { it.run() } }
val producerThread = thread { producerTask.run() }
consumerTasks.forEach { it.running = false }
producerTask.running = false
```

في مرحلة ما في المستقبل، قد نقرِّر إغلاق المنتج والمستهلك، وسنفعل ذلك باستخدام متغيِّر التحكم:

```
consumerTasks.forEach { it.running = false }
producerTask.running = false
```

دعنا الآن نتخيل أن أحد عملائنا كان في الحالة التالية: استدعى take، لكن الطابور كان فارغًا والآن هو في حالة التعطيل، بما أنَّ المُنتِج مغلق الآن، فلن يتلقى المُستهلِك أي عنصر وسيبقى مُعطلًا ومُعطَّلًا، ولأنَّه سيظل كذلك

فلن يتحقق أبدًا من متغير التحكم وبالتالي لن يخرج برنامجنا أبدًا بشكل طبيعي.

لاحظ أنَّه في هذا المثال، ستؤثر هذه المشكلة فقط على المُستهلِك وليس على المُنتِج لأنَّ المُنتِج يُعطَّل لفترة محدودة من الزمن وسيستيقظ في النهاية للتحقق من متغيِّر التحكم.

## مقاطعة الخيوط

لتجنب مثل هذه المشاكل السابقة، يجب علينا إيجاد ما يقاطع عمل الخيط، فالمقاطعة (interrupt) هي طريقة لإيقاظ خيط مُعطَّل حاليًا غصبًا لمتابعة عمله، فهو يقاطع حرفيًا الخيط، وعند حدوث هذا، ترمي الحالة الحاجبة الاستثناء `InterruptedException` والذي يجب التعامل معه وهذا الاستثناء هو طريقك لمعرفة أنَّ الخيط قد جرى مقاطعته أم لا.

لنضيف على المثال السابق مفهوم المقاطعات وذلك إلى المستهلك:

```
class InterruptableConsumerTask(val queue: BlockingQueue<Int>) :  Runnable
{

  override fun run() {
    try {
      while (!Thread.interrupted()) {
        val element = queue.take()
        println("I am processing element $element")
      }
    } catch (e: InterruptedException) {
      // shutting down
    }
  }
}
```

كما ترى، فإنَّ الحلقة مضمنة في كتلة `try...catch`، إذا اشتغلت، وهذا يسمح بتشغيل الدالة بشكل طبيعي، مما يضمن انتهاء الخيط، ولاحظ أنَّ حلقة `while` اللانهائية أصبحت تعليمة `while` مع شرط كذلك. يتحقق الشرط `Thread.interrupted()` ما إذا كان قد تمت مقاطعة الخيط منذ آخر مرة استدعيت فيه الدالة، وهذا مطلوب

لأنَّه إذا لم يحظر الخيط الحالي في `take()` عند حدوث المقاطعة، فلن يـرمى أي اسـتثناء ولن نتمكن من الخـروج، وهذا مهم جدًا عند استخدام المقاطعات للتعامل مع الحالتين.

لتنفيـذ المقاطعـة، نسـتدعي `interrupt` على نسـخة `Thread`، ولـذلك، تحتـاج شـيفرة البرمجيّـة لإيقـاف التشغيل إلى العمل على نسخ الخيط نفسها وليس المهام:

```
val queue = LinkedBlockingDeque<Int>()

val consumerTasks = (1..6).map {
   InterruptableConsumerTask(queue)
}
val producerTask = ProducerTask(queue)

val consumerThreads = consumerTasks.map {
   thread { it.run() }
}
val producerThread = thread { producerTask.run() }
consumerThreads.forEach { it.interrupt() }
producerTask.running = false
```

لاحظ أنَّه بالنسبة إلى المنتج، فلا ننفِّذ المقاطعة حيث تعمل متغيرات التحكم بشكل جيِّد.

## قيد وحدة المعالجة المركزيَّة مقابل قيد الدخل والخرج

أحد المصطلحات المشهورة من مصطلحات عالم الخيوط هو مفهوم حسـابات قيـد وحـدة المعالجـة المركزيّـة (CPU-bound) وقيد الإدخال/الإخراج (I/O-bound)، وهذا يعني ببساطة أن تهيمن مهمة معينة على اسـتخدام CPU أو I/O بغض النظر عما إذا كانت شبكة أو ملـف أو أي شـيء آخـر؛ فعلى سـبيل المثـال، فحسـابات قيـد وحـدة المعالجة المركزيّة (CPU-bound computation) هو الحساب الذي يمكنك من خلاله حساب أرقام Pi، ومثـال على حسابات قيد I/O هو الذي يمكنك تنزيل الملفات منه من الإنترنت وحفظها محليًا.

في المثال الأول، يمكننا تحقيـق تقـدم سـريع بسـرعة واحـدة المعالجـة المركزيـة CPU عنـدما تعـالج العمليـات الرياضية، وفي المثال الثاني، يمكننا إحراز التقدم بسرعة الشبكة التي تزودنا بالبايتات، وستكون الحالة الأخيرة أبطأ

بكثير.

المفهوم مهم عند تقرير كيفية تقسيم عمليات التنفيذ إلى خيوط؛ لنفترض أنه كان لدينا مجمَّع خيوط (thread pool) مكون من 8 خيوط ولقد خصَّصنا هذا المُجمَّع لكل من حسابات قيد وحدة المعالجة المركزيّة وقيد I/O. فإذا كانت هذه هي الحالة، فمن الممكن أن يكون لدينا حالة تُعطَّل فيها الخيوط الثمانية مع شبكة بطيئة منتظرة استلام البايتات منها في حين أن حساب Pi لن يتقدم على الرغم من أنَّ وحدة المعالجة المركزيّة في وضع الخمول.

الحل الشائع لهذا هو أن يكون لديك مجمَّعين من الخيوط، واحد لعمليات قيد وحدة المعالجة المركزيَّة، والذي قد يقتصر حجمها على عدد نوى وحدة المعالجة المركزيَّة، والآخر لعمليات قيد I/O، والذي يكون في العادة أكبر لأنَّ هذه الخيوط غالبًا ما تكون في حالة التعطيل في انتظار البيانات.

## 2. قفل جامد وقف متحرك

عندما يتعذر استمرار عمل خيط لأنَّه يتطلب بعض الموارد التي يمتلكها خيط آخر، فيُعطِّل منتظرًا هذا المورد؛ في المقابل، الخيط الذي يمتلك هذا المورد بدوره يتطلب شيئًا يمتلكه الخيط الأول وبذلك يُعطِّل هو ويُعطِّل ذلك الخيط الآخر أيضًا، ولا يمكن لأي واحد منهما التقدم، وهذا ما يسمى بالقفل الميت (Deadlock) أو القفل الذي لا يُفتَح.

إذا كان المورد يتيح تدخل النظام التشغيل أو الآلة الافتراضيَّة، يمكن أن يُفتَح هذا القفل بإيقاظ أحد الخيطين ومن ثم سيتمكن الخيط آخر من الحصول على المورد، وسيُستَأنف التقدم، وعمليًا فهذه الحالة هي ليست حالة قفل جامد (قفل ميت). أضف إلى ذلك أنَّ الموارد يجب أن لا تكون قابلة للمشاركة، وإلا لما حصلت تلك الحالة وتمكن الخيطان من العمل وبالتالي لن تكون الحالة حالة قفل جامد.

إحدى طرق تجنب حالة القفل الجامد هي التأكد من أنَّ الخيوط تطلب ملكية مورد في نفس الترتيب؛ فإذا كان لديك الخيطان `t1` و `t2` وكلاهما يتطلبان الموردين `r1` و `r2`، وإذا كانا يتطلبان دائما `lock(r1)` ثمَّ `lock(r2)` فإنَّه من المستحيل الوصول إلى حالة يملك فيها أحد الخيطين المورد `r1` و الخيط آخر يملك المورد `r2`، وهذا لأنَّ الخيط الذي سيحصل على ملكية المورد `r1` سيمنع الخيط الآخر من طلب المورد `r2` إلى أن يملك المورد `r1` أولا.

في الجهة المقابلة، حالة القفل الحي (livelock) أو القفل المتحرِّك هو الوضع الذي تكون فيه الخيوط قادرة على تغيير حالتها لكن في النهاية لا تتقدم، فعلى السبيل المثال، إذا كانت لدينا شيفرة برمجيّة تتحقّق من حدوث حالة قفل ميت جامد، وواحدة تجبر الخيطين على فك الأقفال، فيمكن أن نصل إلى حالة تعيد فيها الخيوط طلب الأقفال في نفس الترتيب السابق، والعودة إلى حالة القفل الميت، فعلى الرغم من أنَّ الخيوط تنتقل من حالة التعطيل إلى العمل والعكس، فعلى الأقل تبدو أنَّها تفعل شيئًا ما رغم أنَّها لن تُحقِّق أي تقدم في نهاية المطاف لإكمال حساباتها.

من المهم التفكير في حالتي القفل الميت الجامد والقفل الحي المُتحرِّك عند كتابة شيفرة برمجية متزامنة لضمان صحة البرنامج والأداء، وهذا صحيح بشكل خاص لأنَّ هذه الأنواع من العلل يمكن أن تظهر في بعض الأحيان فقط عند تشغيل برنامجك على أنظمة معيَّنة وتحت ظروف معينة، ولذلك قد تبدو الشيفرة البرمجيَّة صحيحة عندما يكون هنالك خلل خفي.

## أ. مشكلة عشاء الفلاسفة

مشكلة عشاء الفلاسفة (dining philosophers problem) هي مشكلة كلاسيكيَّة في علوم الحاسوب، إذ ظهرت المشكلة على يد إدسجير ديكسترا (Edsger Dijkstra) الشهير بالعديد من المساهمات في تطوير البرمجيات، وهي مشكلة تستخدم لإظهار كيف يمكن أن تؤدي مشكلات المزامنة إلى حالة توقف تام (أقفال جامدة) وتلك التي لها حل ليست بسيطة دائمًا.

المشكلة في شكلها الحالي تشبه هذا: تخيَّل أن طاولة من خمسة فلاسفة، كل واحد يجلس أمام وعاء من السباغيتي، بين كل فيلسوف شوكة، حيث لكل واحد منهم حق الوصول إلى شوكتين، واحدة على كل جانب منه. يمكن للفيلسوف التفكير والأكل والتنقل بين هاتين الحالتين عشوائيًا. ومن أجل تناول الطعام، يجب عليه أو عليها أن يحصل على الشوكتين في نفس الوقت، وإذا كانت شوكةٌ غير متوفِّرة، أي يستخدمها فيلسوف آخر، فسينتظر هذا الفيلسوف حتى تكون متاحة، ليحمل الأخرى حالما تصبح متاحة. نفترض أنَّ الصحن لن يفرغ أبدًا وأن الفلاسفة سيكونون جائعين دومًا.

لإظهار أنَّ الحل الواضح سيؤدي إلى حالة توقف تام بقفل ميت، ضع في اعتبارك الأخطاء التالية في الحال:

يجب على كل فيلسوف:

- التفكير لفترة عشوائية من الزمن.
- محاولة الحصول على الشوكة اليسرى، والتعطيل حتى تكون متاحة.
- محاولة الحصول على الشوكة اليمنى، والتعطيل حتى تكون متاحة.
- تناول الطعام لفترة زمنيَّة عشوائية.
- تحرير كلا الشوكتين.
- تكرار العملية.

هذا خطأ لأنَّه من السهل الدخول في حالة يملك فيها كل فيلسوف شوكته اليسرى مما يعني أنَّه لا يمكن للفلاسفة الحصول على الشوكة اليمنى مطلقًا (لأنَّ كل شوكة يمنى لأحد الفلاسفة هي الشوكة اليسرى لفيلسوف آخر).

يمكن استخدام المشكلة كذلك مثالًا على حالة قفل حي متحرك (livelock)؛ تخيّل معي أنَّنا نعزز الحل بقاعدة أخرى: إذا تعطلت عملية الحصول على شوكة لأكثر من دقيقة، ينبغي إسقاط جميع الشوك وإعادة تشغيل الإجراء، ففي هذه الحالة لا يمكن حصول حالة قفل ميت جامد، ويمكن للنظام آنذاك إحراز تقدم دائمًا (من معطل إلى عامل)، ومع ذلك، فمن الممكن أيضًا حصول حالة يلقي فيها كل الفلاسفة الأشواك في نفس الوقت، مما يعني أنَّهم سيستمرون في العودة إلى حالة التعطيل (blocking state).

## 3. المنفِّذون

إن إنشاء خيط يدويًا جيد عندما نرغب في بأن ينجز خيط واحد بعض العمل، وربما خيط طويل العمر أو مهمة لمرة واحدة بسيطة للغاية ستعمل في نفس الوقت، ومع ذلك، عندما نريد تشغيل العديد من المهام المختلفة تزامنيًا أثناء مشاركة وقت وحدة المعالجة المركزيّة المحدود، تتبع عملية المهام بطريقة سهلة، أو عليك ببساطة تجريد (abstract) عمل كل مهمة، وهنا يمكننا الانتقال إلى `ExecutorService`، ويسمى هذا «منفذًا» (executor) كذلك، وهو جزء من مكتبة جافا القياسيَّة.

**ملاحظة**

المنفذ (executor) هو أكثر من واجهة مُعمَّمَة الأنواع (generic interface) منه من الدالة `run()` الوحيدة. الواجهة `ExecutorService` هي واجهة أكثر تميزًا وعادة ما تستخدم التجريد (abstraction)، ومن الشائع بالنسبة للناس استخدام مصطلح المُنفِّذ عند الإشارة إلى أي واحد منهما.

إنَّ `ExecutorService` هو ببساطة كائن ينفِّذ المهام المرسلة بينما يسمح لنا بالسيطرة على دورة حياة المُنفِّذ، أي رفض المهام الجديدة أو مقاطعة المهام العاملة، كما يسمح لنا المنفذون بتجريد آلية تخصيص الخيوط إلى المهام، مثلًا قد يكون لدينا منفذ مع عدد محدد من الخيوط أو منفذ ينشئ خيطًا جديدًا لكل مهمة يرسلها. وأي مهمة لا تنفذ حاليًا ستوضع في الطابور داخليًا في المُنفِّذ.

يعمل المنفذون بواجهتين رئيسيتين، الأولى `Runnable` وهي الأكثر تعميمًا (generic) وتستخدم الواجهة عندما نريد فقط أن نُغلِّف بعض التعليمات البرمجية لتعمل في مُنفِّذ، الثانية وهي `Callable` تضيف قيمة إرجاع عند اكتمال المهمة. لمَّا كانت كل واجهة منهما هي واجهة ذات تابع-مجرَّد-مفرد (single-abstract-method interface)، فيمكننا تمرير دالة مُجرَّدة (function literal) فقط في كوتلن.

تأتي مكتبة جافا القياسيّة مع عدد من المنفذين الضمنيين (built-in executors)، إذ تم إنشاءها من توابع مساعدة في `Executors`، والتي تسمح لك بإنشاء مُنفِّذ مخصص بسهولة. أكثر المُنفِّذين المستخدمين هما `Executors.newSingleThreadExecutor()` والذي ينشئ منفذًا يعالج مهمة واحدة في وقت واحد و `Executors.newFixedThreadPool(n)` الذي يُنشئ منفذًا مع مجمَّع خيوط داخلي يُشغِّل مهامًا تصل إلى `n` مهمة بشكل متزامن.

دعنا نرى كيف يمكننا التعامل مع دورة حياة المنفِّذ:

```
val executor = Executors.newFixedThreadPool(4)
for (k in 1..10) {
  executor.submit {
    println("Processing element $k on thread ${Thread.currentThread()}")
    }
}
```

في هذا المثال، أنشأنا مُجمِّع خيوط من أربعة خيوط ومن ثم أرسلنا 10 مهام، يجب على كل مهمة طباعة معرّف الخيط التي تشغله. التابع الساكن `Thread.currentThread()` يرجع الخيط الذي تُنفَّذ تعليماته البرمجيّة حاليًا، ويجب أن تبدو المخرجات كالتالي:

```
Processing element 2 on thread Thread[pool-1-thread-2,5,main]
Processing element 5 on thread Thread[pool-1-thread-2,5,main]
Processing element 1 on thread Thread[pool-1-thread-1,5,main]
Processing element 7 on thread Thread[pool-1-thread-1,5,main]
Processing element 8 on thread Thread[pool-1-thread-1,5,main]
Processing element 9 on thread Thread[pool-1-thread-1,5,main]
Processing element 10 on thread Thread[pool-1-thread-1,5,main]
Processing element 3 on thread Thread[pool-1-thread-3,5,main]
Processing element 4 on thread Thread[pool-1-thread-4,5,main]
Processing element 6 on thread Thread[pool-1-thread-2,5,main]
```

لن تكون بنفس الترتيب بالضبط، لأنَّ الناتج غير محدَّد، ويبيِّن هذا كيف تتداخل المهام المختلفة:

```
executor.shutdown()
executor.awaitTermination(1, TimeUnit.MINUTES)
```

بمجرد الانتهاء من المثال، نستدعي `shutdown()` بحيث يمكن رفض المهام الإضافيّة ومن ثم استخدام `await()`، الذي من شأنه أن يحجب البرنامج حتى ينتهي المنفذ من تنفيذ جميع المهام، إذا كنا نريد إلغاء مهام التشغيل، فيمكننا استخدام دالة `shutdownNow()` على المنفّذ، والتي سترفض المزيد من المهام وتقاطع تشغيل المهام قبل إعادتها.

## 4. حالات التسابق

حالة التسابق (race condition) هو نوع آخر من أخطاء التزامن التي تحدث عند يصل خيطين أو أكثر إلى بيانات مشاركة بينهما ومحاولة تغييرها في نفس الوقت، وهذا يعني الحالة التي تتطلب فيها مخرجات جزء منطقي تنفيذ تلك الشيفرة البرمجية المتداخلة أو المشتركة بترتيب معيّن، الأمر الذي لا يمكن ضمانه.

مثال كلاسيكي على ذلك هو حساب مصرفي، حيث يضع الخيط الأول المال في الحساب والثاني يخصم منه. تتطلّب عملية الحساب منا استرداد قيمة الحساب، وتحديثها وتخزينها مرَّة أخرى، مما يعني أن ترتيب هذه التعليمات يمكن أن يتداخل مع بعض.

فعلى سبيل المثال، افترض أن حسابًا يحوي 100 دولارًا، ومن ثم نرغب بإضافة 50 دولارًا وسحب 100 دولار منه، فيمكن أن يكون أحد هذه الترتيبات شيئًا كهذا:

| خيط الدائن <credit thread> | رصيد الحساب <account balance> | خيط المدين <debit thread> |
|---|---|---|
| | قيمة البداية = 100 | |
| | | الحصول على الرصيد الحالي = 100 |
| الحصول على الرصيد الحالي = 100 | | |
| | | ضبط الرصيد الجديد: 100 + 50 = 150 |
| | تحديث إلى 150 | |
| تعيين الرصيد الجديد= 100 - 100 | | |
| | تحديث إلى 0 | |

كما ترى، فإن العميل قد فقد إيداعه (لن يشعروا بالقلق إذا خسروا السحب :-D ).

يمكن أن يختلف الترتيب الفعلي في كل مرة نديرها، هذا لأنَّه إذا كان كل خيط يعمل على معالج منفصل، فإن التوقيتات لن تكون متزامنة تمامًا، وإذا كانت الخيوط تعمل على نفس النواة، فإنه لا يمكننا أن نكون على يقين من

فترة الخيط قبل حدوث تبديل السياق.

واحدة من القضايا الخاصة مع حالات التسابق هو أنَّها بطبيعتها قد لا تكون واضحة على الفور، وهذا يعني أنَّها غير حتميَّة الحدوث، فالآلة التي تستخدم للتطوير سيكون لديها سرعات معالجة مختلفة عن الخادم الذي سيُستعمَل في بيئة الإنتاج، أو عدد المستخدمين المتزامنين، وهذا قد يكون كافيًا لبدء حالة سباق قد لا تراها في بيئة التطوير.

## أ. المراقبون

في JVM، تملك جميع النُسخ ما يعرف بالمراقب (Monitor)، فيمكن عدُّه بمثابة رمز خاص والذي يسمح لخيط واحد فقط بامتلاكه في أي لحظة معينة، يمكن لأي خيط أن يطلب مراقبًا لأي نسخة، وفي هذه الحالة سيستلمه أو يُعطّله حتى تقديم الطلب، وبمجرَّد أن يمتلك الخيط لمراقب معيَّن، فيقال أنَّه يملك المراقب (hold the monitor).

لطلب المراقب، نستخدم الدالة `synchronized` والتي تعد في كوتلن إحدى دوال المكتبة القياسيّة بدلًا من ميزة مدمجة كما هو الحال في جافا؛ تقبل هذه الدالة معاملين: الأول هو الكائن الذي نرغب في منحه مراقبًا والثاني هو دالة مُجرَّدة، والتي ستنفّذ بمجرَّد تعيين المراقب. إليك الشيفرة البرمجيّة التالية:

```
val obj = Any()
synchronized(obj) {
   println("I hold the monitor for $obj")
}
```

إذا فحصنا البايتكود لهذا، فسنرى أنَّنا نحصل على المراقب (monitorenter) ونطلق سراحه (monitorexit):

```
0: new
3: dup
4: invokespecial
7: astore_0
9: aload_0
10: monitorenter
13: getstatic
```

```
16: astore_2
17: aload_0
18: monitorexit
19: aload_2
20: goto
23: astore_2
24: aload_0
25: monitorexit
26: aload_2
27: athrow
28: pop
29: return
```

يُضمَن لأي شيفرة برمجيَّة تُنفَّذ عندما تكون داخل المراقب أن تُنفَّذ بالكامل (سواء بشكل عادي أو عن طريق رمي استثناء) قبل تحرير المراقب وقبل أن يحصل أي خيط على ملكية هذا المراقب، والشيفرة البرمجية التي نشغلها مع الاحتفاظ بمراقب يشار إليها على أنَّها قسم حرج (critical section).

عندما يصل الخيط إلى استدعاء متزامن لمراقب مملوك بالفعل بواسطة خيط آخر، فسيوضع في مجموعة من الخيوط المنتظرة (في حالة الانتظار). وبمجرَّد أن يترك الخيط المراقب، سيُختَار أحد الخيوط المنتظرة لامتلاكه، ولا يوجد أي ضمان بشأن أي الخيوط سيحصل على المراقب أولًا، فالخيط الذي يأتي أولًا ليس له أولوية على الذي يصل آخرًا.

الاستخدام الرئيسي للكتلة المتزامنة (synchronized block) هو التأكد أنَّ خيط واحد فقط يمكنه تغيير المتغيرات المشاركة في نفس الوقت. إذا عدنا لمثال الحساب البنكي وهذه المرة لتحديثه لاستخدام التزامن في بعض الحالات الشائعة، فسنرى فرقًا في تداخل الشيفرة البرمجيَّة:

| خيط الدائن<br><credit thread> | رصيد الحساب<br><account balance> | خيط المدين<br><debit thread> |
|---|---|---|
| | قيمة البداية = 100 | |
| طلب المراقب للحساب | | |

| | | طلب المراقب للحساب |
|---|---|---|
| الحصول على المراقب | | |
| الحصول على الرصيد الحالي = 100 | | |
| رصيد جديد = 100 +50 | | |
| | تحديث إلى 150 | |
| تحرير المراقب | | |
| | | الحصول على المراقب |
| | | الحصول على الرصيد الحالي = 150 |
| | | رصيد جديد = 150 +50 |
| | تحديث إلى 200 | |
| | | تحرير المراقب |

للتوضيح، لا تعمل المزامنة كتقنيـة إلا إذا كـانت الخيـوط تطلب مراقبًا لنفس النسـخة بالضـبط؛ كـل نسـخة من صنف لها مراقب خاص بها ولذلك لا يستفيد وجود خيطين من نسخ مختلفـة من نفس الصـنف، وهـذا أحـد الأخطـاء الشائعة للمبتدئين.

التزامن هي تقنية متقدمة إلى حد ما، إذ تستخدم عادة للمزامنة عبر مجموعة كبيرة نسـبيًا من التعليمـات الـتي تعطّل خيوط أخرى لفترة طويلة، ونحن نسعى لتحقيق أكبر قدر من الإنتاجيـة في الشـيفرة البرمجيّـة عـبر الـتزامن، ويجب علينا أن نحاول تقليل مقدار الوقت الذي نقضيه في أي قسم حرج من الشيفرة.

## ب. الأقفال

بديل التزامن هو استخدام إحدى تنفيذات القفل الموجودة في حزمة `java.util.concurrent.locks`، وعادةً، التنفيذ هو `ReentrantLock`، فقفل إعادة الدخول (reentrant lock) هو الذي يسمح لمالك القفل الحالي أن يطلب القفل مرَّة أخرى دون التسبب حالة قفل ميت جامد، وهذا يبسط التعليمات التي تستخدم العودية أو تمرير القفل إلى دوال أخرى.

رغم تشابه استخدامات الأقفال (locks) والمزامنة (synchronization) إلى حد بعيد، مثل تقييد الوصول إلى كتلة من التعليمات البرمجيّة، فإن واجهة القفل أكثر قوَّة؛ على سبيل المثال، يسمح لنا القفل بالحصول على الملكية ثم التراجع إذا لم تكن ناجحة، بينما يعطّل الاستدعاء المتزامن فقط.

في المثال التالي، إذا لم نحصل على lock على الفور، فسنستمر؛ تشير قيمة العودة لدالة `tryLock()` ما إذا حصلنا على القفل أم لا:

```
val lock = ReentrantLock()
  if (lock.tryLock()) {
    println("I have the lock")
    lock.unlock()
  } else {
    println("I do not have the lock")
  }
```

تذكَّر أن تحرر `lock` بعد استخدامه، فيمكن `lock` أن يُعطّل أيضًا مع السماح لك بمقاطعته:

```
val lock = ReentrantLock()
try {
  lock.lockInterruptibly()
  println("I have the lock")
  lock.unlock()
} catch (e: InterruptedException) {
  println("I was interrupted")
}
```

توفّر كوتلن دالة مُوسّعة تسمح لنا باستخدام `lock` وتحريره تلقائيًا:

```
val lock = ReentrantLock()
lock.withLock {
  println("I have the lock")
}
```

ميّزة أخرى هي أنَّ القفل يسمح لنا بفرض الطلب العادل (fair ordering)، مما يضمن عدم وجود خيط يموت أثناء انتظار فتح القفل، ويتم ذلك عن طريق تخصيص القفل للخيط الذي كان ينتظر لأطول فترة، ولكن يمكن لهذا أن يكون له مساوئ سلبية على الأداء، لا سيما مع أقفال متنافسة للغاية (highly contended locks).

> **ملاحظة**
>
> التنافس (Contention) هو المصطلح المعطى لمقدار الطلب على القفل أو المراقب، ويعني وجود قدر كبير من التنازع أنَّ العديد من الخيوط تتنافس على نفس القفل في نفس الوقت.

## أقفال القراءة والكتابة

نوع أكثر تعقيدًا من الأقفال الذي توفره المكتبة القياسية هو `ReadWriteLock`، وهذا قفل مخصّص للمشاكل التي تشمل تجميعات من القُرّاء (readers) والكُتّاب (writers). تخيل معي برنامجًا يقرأ البيانات من الملف وأحيانًا يحدّث الملف، وهو آمن تمامًا لأن تقرأه عدة خيوط دفعةً واحدةً ولكن فقط طالما لم يتم تعديل الملف من قبل أي شخص؛ بالإضافة إلى ذلك، يجب أن يكتب كاتب واحد في كل مرة على الملف منعًا لتعديله في الوقت نفسه من أكثر من كاتب.

لتحقيق هذا، يملك قفل القراءة والكتابة (read-write lock) نوعين من الأقفال: قفل القراءة (read lock) وقفل الكتابة (write lock). يمكن طلب قفل القراءة بواسطة خيوط متعدّدة وأما قفل الكتابة فلا يمكن إلا لخيط واحد الاحتفاظ به فقط. إذا كان هنالك من يمتلك قفل القراءة، فلا يمكن الحصول على قفل الكتابة، وبمجرّد الحصول على قفل الكتابة، فلا يمكن للخيوط الأخرى الحصول عليه أو على قفل القراءة حتى تحريره.

يجب أن يُؤخَذ التصميم الأساسي لقفل القراءة والكتابة في الحسبان أيضًا ما إذا كان لقارئ ثانٍ يطلب قفل

القراءة الأفضلية على الذي ينتظر قفل الكتابة، ولشرح هذا، تخيَّل أنَّ الخيط الأول يحتفظ بقفل القراءة والخيط الثاني يطلب قفل الكتابة. فعلى الرغم من أنَّ الخيط الثاني ينتظر القارئ الأول أن ينتهي، فقد يأتي قارئ آخر ويطلب قفل القراءة، فلمن يجب تخصيص ذلك القفل إذا؟ فلا مشكلة من الحصول على قرَّاء متعددين، ولكن ماذا لو بقي هذا لأجل غير مسمى؟ من المؤكد أن الكاتب سيموت منتظرًا دوره.

لتجنب هذا، يمكننا إنشاء قفل القراءة والكتابة في الوضع العادل على غرار تنفيذات القفل القياسي؛ ففي الوضع العادل، الكاتب الذي كان ينتظر أطول فترة سيحصل على قفل الكتابة، وإذا كان القارئ ينتظر لفترة أطول، فسيُعطَى إلى جميع القراء الذين ينتظرون القفل في نفس الوقت.

## ت. الوصول الحصري

ابتكر صديقنا القديم إدسجير ديكسترا (Edsger Dijkstra) مفهوم الوصول الحصري (Semaphore، أو الكائن حصري الوصول)، على الرغم من أنَّه هذه الأيام قد لا تستخدم هذا المفهوم بالقدر الذي كان عليه سابقًا مع لغات برمجة عالية المستوى، فإنه من المفيد فهم ماهيته وفائدته، وهذا لأن مفهوم الوصول الحصري يُستخدَم عادة كأساس للتجريدات على مستوى أعلى (higher level abstractions).

فالوصول الحصري هو آلية تحتفظ بعدد الموارد وتسمح بتغيير عداد (counter) بطريقة آمنة: إمَّا لطلب الموارد أو إعادتها، مع القدرة الإضافية على الانتظار اختياريًا إلى أن يكون العدد المطلوب من الموارد متاحًا؛ في التصميم الأصلي، سميت عملية طلب مورد باسم p وعملية إرجاع المورد باسم v، وتأتي الحروف هذه من كلمات هولنديَّة، لأن صديقنا إدسجير كان هولنديًا.

تعرض مكتبة جافا القياسيَّة تنفيذ مفهوم الوصول الحصري في الصنف `java.util.concurrent.Semaphore`. في مصطلحات جافا، يدعى التعداد بعدد التصريحات (number of permits) وتدعى p أو up بالطلب (acquire) ويدعى v أو down بالتحرير أو الإفلات (release).

الفائدة من الوصول الحصري ليس فقط أنه يمكن استخدام الموارد بأمان من قبل خيوط متعدِّدة دون الوقوع في حالة تسابق، ولكن إن انتظار أي خيط لعملية طلب المورد سيؤدي إلى التعطيل، وبذلك نتجنب الحاجة إلى محاولة فتح القفل وإهدار وقت وحدة المعالجة المركزيّة.

محاولة فتح القفل (spin lock) هي نوع من أنواع الأقفال إذ يختبر الخيط بشكل متكرّر تحقق شرطٍ ليكمل عمله، وبما أن الخيط نشط، فسيهدر وقت المعالج دون تنفيذ أي عملية مفيدة، وهذا مثال ما يسمى بعملية الانشغال بالانتظار (busy-waiting) وهو حل أدنى لتعطيل خيط بشكل صحيح.

حالة خاصة من الوصول الحصري هو ما يسمى الوصول الحصري الثنائي (binary semaphore)، الذي يحتوي على مورد واحد لذلك فهو يمتلك الحالتين 0 و 1 أو مقفل وغير مقفل. ويمكن استخدام هذه لتنفيذ قفل أو تقييد وصول المورد إلى مستهلك واحد في أي لحظة.

## مشكلة المخزن المؤقّت المُقيّد

مشكلة المخزن المؤقت المُقيّد (bounded buffer) أو المنتج-المستهلك (producer-consumer) هي مشكلة كلاسيكيّة في التزامن، فهذه المشكلة هي كالتالي: وجود مُنتِج من شأنه أن يولّد العناصر التي يمكن وضعها في مخزن مؤقت ثابت الحجم ويوجد مستهلك يقرأ هذه العناصر. يجب أن لا يحاول المُنتِج توليد العناصر إذا كان المخزن ممتلئًا، ويجب ألا يحاول المستهلك قراءة العناصر إذا كان المخزن فارغًا.

ستشبه محاولة كتابة شيفرة أولية ساذجة دون استخدام أنواع أساسية متزامنة مثل الشيفرة التالية:

```
val buffer = mutableListOf<Int>()
val maxSize = 8

(1..2).forEach {
   thread {
      val random = Random()
      while (true) {
         if (buffer.size < maxSize)
            buffer.plus(random.nextInt())
      }
   }
}
(1..2).forEach {
```

```
        thread {
            while (true) {
                if (buffer.size > 0) {
                    val item = buffer.remove(0)
                    println("Consumed item $item")
                }
            }
        }
    }
```

يوجد مخزن مشترك مع مُنتجَيْن ومُستهلكَيْن كل واحد منهما يصل إلى الآخر، ويتحقق المنتجان والمستهلكان على التوالي ما إذا كان هنالك مساحة لإنتاج عنصر أو استهلاك عنصر، ويفعلان ذلك بمجرَّد التحقق من حجم القائمة التي تمثِّل المخزن، والمشكلة مع هذا الحل هو أننا نقوم ندور القفل محاولين فتحه بانتظار العنصر في كل مرة، وإذا كان المخزن فارغًا، فسيستمر المستهلك في التحقق من الشرط وإهدار وقت وحدة المعالجة المركزيّة.

لذلك، نحن بحاجة إلى كتابة تنفيذ آخر، لأنَّ لدينا عددًا من الأماكن في المخزن، فعلى ما يبدو أن مفهوم الوصول الحصري جيد هنا، ويرجع هذا إلى القدرة على الاحتفاظ بالعدد (count)؛ والفكرة وراء التكرار التالي هو أن لدينا وصولين حصريين: يحتوي الأول على الأماكن الفارغة والثاني على الأماكن الممتلئة في المخزن. سينتظر المُنتِج مكانًا فارغًا قبل أن ينتج أي عنصر، وإن أنتج عنصرًا فسيزيد عدد الأماكن المملوءة، وسينتظر المستهلك بدوره مكانًا ممتلئًا قبل استهلاك أي عنصر، وبعد هذا، سيزيد عدد الأماكن الشاغرة:

```
val emptyCount = Semaphore(8)
val fillCount = Semaphore(0)
val buffer = mutableSetOf<Int>()

thread {
    val random = Random()
    while (true) {
        emptyCount.acquire()
        buffer.plus(random.nextInt())
```

```
      fillCount.release()
    }
  }

  thread {
    while (true) {
      fillCount.acquire()
      val item = buffer.remove(0)
      println("Consumed item $item")
      emptyCount.release()
    }
  }
```

هذا تحسين جيّد بالتأكيد ويتجنّب الوقوع بأي حالة قفل، ومع ذلك، بما أنَّه لا يزال بإمكان عدة خيوط الوصول إلى القائمة في وقت واحد، فيمكن تعديل القائمة من قبل أكثر من خيط في نفس الوقت، ويمكننا أن نرى ذلك من خلال جدول التعليمات الذي يوضّح تداخلًا محتملًا للتعليمات:

| عملية أولى<br><producer 1> | القائمة التي تمثِّل المخزن<br><list> | عملية ثانية<br><producer 2> |
|---|---|---|
| | الحجم = 6 | |
| طلب مكان شاغر | | |
| | | طلب مكان شاغر |
| أخذ مكان شاغر | | |
| | | أخذ مكان شاغر |
| تعيين المكان الشاغر 7 إلى x | | |
| | الحجم = 7 | |

| | | |
|---|---|---|
| | | تعيين المكان الشاغر 7 إلى y |
| | الحجم = 7 | |

تحدث هذه المشكلة بسبب أن الخيوط المتعدّدة ستغيّر القائمة داخليًا في نفس الوقت، وإن تحديث القائمة لا يجري بضربة واحدة ويتطلّب العديد من التعليمات، والتي تخضع بأنفسها لحالات التسابق.

**ملاحظة**: يقال أنّ العملية تكون أحادية التنفيذ (atomic) إذا ظهرت لبقية النظام كعمليّة واحدة وأي حالة وسيطة لن تكون مرئيّة أبدًا خارج الخيط.

لذلك فإن الحل الآمن هو زيادة تقييد الوصول إلى القائمة لخيط واحد في كل مرّة، ويمكننا القيام بذلك عن طريق تقديم `mutex`:

```
val emptyCount = Semaphore(8)
val fillCount = Semaphore(0)
val mutex = Semaphore(1)
val buffer = mutableSetOf<Int>()

thread {
   val random = Random()
   while (true) {
      emptyCount.acquire()
      mutex.acquire()
      buffer.plus(random.nextInt())
      mutex.release()
      fillCount.release()
   }
}

thread {
   while (true) {
```

```
        fillCount.acquire()
        mutex.acquire()
        val item = buffer.remove(0)
        mutex.release()
        println("Consumed item $item")
        emptyCount.release()
    }
}
```

في التكرار النهائي، قمنا بإضافة `mutex` لمسك (acquire) وتحرير (release) كل تغيّر يجرى على المخزن، وهذا الحل هو آمن للخيوط.

## ث. التجميعات المتزامنة

كما هو موضَّح في القسم الخاص بحالات التسابق، فيمكن أن يؤدي وصول عدة خيوط إلى البيانات المشتركة في حالة غير متناسقة، كما رأينا في القسم الخاص بالمراقبين والقفل، فإن كتابة شيفرة برمجيّة آمنة للخيوط لتحديث التجميعات قد يكون خادعًا، ولحسن الحظ، مكتبة جافا القياسية قد حلَّت العديد من هذه المشاكل لنا. ففي الإصدار 1.5 من جافا (أو النسخة 5) وما بعده، تحتوي المكتبة القياسيَّة على عدد كبير من الأنواع الأساسية المتزامنة (concurrency primitives) والتجميعات المتزامنة (concurrent collections).

ستغطي الأقسام العديد التالية بعضًا من هذه الأنواع الأساسية، وسنركّز في بقية هذا الفصل على التجميعات على وجه التحديد وثلاثة أنواع أساسية أخرى لا علاقة لها بالتجميعات.

التجميعة المتزامنة (concurrent collection) هو المصطلح الذي يطلق على التجميعات الآمنة الخيوط (thread-safe) ومصممة على وجه الخصوص للاستخدام في التعليمات البرمجيَّة ذات الخيوط المتعدّدة، فهي أقل في الأداء من التجميعة العاديّة في بيئة الخيط الواحد، لكنَّها أفضل أداءً من تغليف التجميعات في كتل متزامنة (والذي كان الحل قبل إصدار جافا 1.5).

### ConcurrentHashMap

أول تجميعة هي `java.util.concurrent.ConcurrentHashMap` وربما هي الأكثر استخدامًا بين جميع التجميعات المتزامنة؛ كما يوحي الاسم، هذه التجميعة هي تنفيذ للواجهة Map آمنة الخيوط، فالمشكلة مع الخريطة العادية هي أن خيطين قد يحاولان وضع عنصر فيها، فقد يكتب أحدهما فوق الآخر إذا كلا المفتاحين اللذين يملكانهما متماثلين تمامًا. إن المشكلة الأخرى الأقل وضوحًا هي أنه إذا وصلت خريطة لأقصى سعتها مع الخيط الأول، فإنها ستنفِّذ عملية تغيير الحجم، والتي ستنطوي على إعادة كل عنصر في دلو جديد، وفي أثناء ذلك، فإنَّ عملية الإضافة من الخيط الثاني قد تضيع.

تتجنب التجميعة `ConcurrentHashMap` هذه المشكلات، فهي تحافظ على مجموعة من الأقفال، ويُستخدم كل قفل لتقييد الوصول إلى جزءٍ من الخريطة (stripe of the map)، وبهذه الطريقة، يمكن أن تحدث تحديثات متعدِّدة في الوقت نفسه بأمان، والحد من كمية التعليمات البرمجيّة التي يجب تنفيذها بشكل متسلسل، بالإضافة إلى ذلك، لا تتطلّب عملية الجلب `get()` قفلًا على الإطلاق، وسترجع نتيجة التحديث الأحدث.

### الطابور المعطِّل

الطابور المُعطِّل (blocking queue) هو تجميعة أخرى خاصَّة، فهي امتداد لواجهة `java.util.Queue` لدعم عمليات تعطيل الخيوط الآمنة (thread-safe blocking operations)، فهي تعرِّف عملية تسمى `take()`، والتي ستُعطِّل حتى يكون الطابور غير فارغ، والعملية `put()` التي ستمنع التعطيل حتى يكون هنالك مجال في الطابور لقبول عنصر؛ إذا كانت عدة خيوط مُعطَّلة بسبب نفس العملية، لنقل أن ثلاثة خيوط تحاول أن تأخذ عنصرًا واحدًا عندما يُصبح متاحًا، فسينجح خيط واحد فقط وسيبقى الآخرون في حالة تعطيل آمنة.

يوجد تنفيذان في مكتبة جافا القياسيَّة، الأول وهو `java.util.concurrent.ArrayListBlockingQueue` مدعوم من قبل تنفيذ `Array` والثاني وهو `java.util.concurrent.LinkedBlockingQueue` مدعوم من قبل `LinkedList`، لكل واحدة منهما مميزاتها بالطبع، وهذه الأخيرة مفيدة بشكل خاص لاستخدامها قفلين داخليًا، واحد لرأس القائمة والثانية للذيل.

سيؤدي استخدام طابور الحظر إلى تبسيط مشكلة المخزن المؤقت التي رأيناها في وقت سابق، دعنا نعيد صياغة هذه المشكلة باستخدام `LinkedBlockingQueue` حتى نتمكن من رؤية الفرق:

```
val buffer = LinkedBlockingQueue<Int>()

thread {
   val random = Random()
   while (true) {
      buffer.put(random.nextInt())
   }
}

thread {
   while (true) {
      val item = buffer.take(0)
      println("Consumed item $item")
   }
}
```

كما ترى، أُخفيَت جميع التعقيدات المتعلقة بالتزامن من أجلنا، يمكننا استخدام الطابور كما لو كنا في بيئة خيط واحد:

```
13.7.x
```

### ج. المتغيرات الفردية

سنحتاج في كثير من الأحيان إلى قيمة واحدة يمكننا تحديثها بشكل أحادي (atomically) بين الخيوط، فالمجموعة تبدو أمرًا مبالغًا فيه لهذا الغرض وربما أبطأ لذلك الغرض متغير خاص من نوع أساسي. توفّر لنا المكتبة القياسية مثل هذه الأنواع الأساسية في الحزمة `java.util.concurrent.atomic`.

هنالك فرق في التنفيذات لكل نوع أساسي، بالإضافة إلى تنفيذ لمراجعة الكائن (object references)؛ على سبيل المثال، يحتوي `AtomicLong` على عداد `Long` ويوفّر عمليات جلب القيمة الحالية أو تحديث القيمة بطريقة آمنة للخيط. حالة الاستخدام النموذجيّة هي عداد مشترك بين الخيوط، ربما كمولّد ID متزايد:

```
val counter = AtomicLong(0)
```

```
(1..8).forEach {
   thread {
      while (true) {
         val id = counter.incrementAndGet()
         println("Creating item with id $id")
      }
   }
}
```

**ملاحظة**

إذا كنت تستخدم الإصدار 1.8 من JDK أو إصدارًا أحدث، فستشحن مع نوعين أساسيين هما LongAdder و DoubleAdder، وهما أكثر فاعليّة لتلخيص القيم مع وجود عيب في الثبات.

الصنف AtomicReference هو صنف مشابه، لكن بدلًا من عدد، فهو يسمح بأي نوع من المراجع، وهو مفيد للسماح لعدّة خيوط بمشاركة كائن واحد والسماح لها جميعها بتحديث الكائن بشكل آمن، إحدى حالات الاستخدام هذه هي التهيئة الكسولة (lazy initialization) بين الخيوط، فالقيمة الافتراضية هي null وكل خيط يتحقق من منها وإذا وجدها فسيحدث القيمة إلى القيمة الصحيحة:

```
val ref = AtomicReference<Connection>()
(1..8).forEach {
   thread {
      ref.compareAndSet(null, openConnection())
      val conn = ref.get()
    }
}
```

الآن خيط واحد فقط سيستدعي الدالة openConnection()، وسيحدث هذا بتكاسل في المرة الأول عند تنفيذ الخيط.

### ح. الكائن CountDownLatch

الكائن `CountDownLatch` هو عبارة عن نوع أساسي متزامن (concurrency primitive) موجود في جافا منذ الإصدار 1.5 (أو النسخة 5، بناءً على نظام ترقيم جافا الذي تستخدمه). الفكرة الأساسية للمزلاج (latch) هي السماح لخيط واحد أو أكثر بالتعطيل حتى تحرير المزلاج، يمكنك أن تتخيّل أن التسمية تأتي من المزلاج الذي نراه على البوابة - بمجرَّد فتح البوابة بتحريك المزلاج، فيمكن للأغنام خلف البوابة الفرار، وهكذا بالمثل، فإنَّ الخيوط تصطف خلف البوابة، وبمجرَّد تحريك المزلاج وتحريره، يُسمَح للخيوط بالتحرك عبرها.

يُهيَّأ المزلاج بعداد، ويمكن استخدام التابع `countDown()` لإنقاص العداد، وبمجرَّد أن يصل العداد إلى الصفر، يلغى تعطيل الخيوط التي تنتظر فتح المزلاج؛ يمكن للخيط تعطيل المزلاج باستخدام التابع `await` وفي الحقيقة يمكن لأي عدد من الخيوط تعطيل المزلاج وسيتحررون جميعًا في نفس الوقت آنذاك.

| ملاحظة | أي خيط يستدعي `countDown` يمكنه الاستمرار بحريَّة، فقط الخيوط التي تستدعي `await` تُعطَّل، ولاحظ أيضًا أنَّه يمكن لأي خيط استدعاء `countDown` عدَّة مرات، وهو ما يحدث غالبًا عندما يكون لدينا العديد من المهام التي تعالج من خلال خيوط متعددَّة. |
|---|---|

للمزالج العديد من الاستخدامات، ذكرنا واحدًا باختصار في الفصل الحادي عشر، الاختبار في كوتلن، عندما أظهرنا أن المزالج هي أداة مفيدة لاختبار الدوال غير المتزامنة، وتذكر أنَّنا رغبنا بمنع التأكيدات من تنفيذها حتى ينتهي تنفيذ الشيفرة البرمجية غير المتزامنة التي تعتمد عليها.

ومن الاستخدامات الأخرى للمزالج هي منع بعض الخيوط الرئيسية من المتابعة حتى استخدام الخيوط العاملة؛ لنفترض أن لدينا تطبيقًا يحتاج إلى تنزيل ومعالجة مصادر تغذية feed مختلفة قبل إرسال تنبيه عبر الطابور، فنحن نرغب في تعدد خيوط معالجة التغذية لاسيما وأنها مقيّدة بوحدة المعالجة المركزيّة (CPU) ونحن نعمل على معالج متعدّد النواة، فيجب إرسال الإشعار الأخير عندما تنتهي معالجة جميع التغذيات؛ نحن لا نعرف مقدمًا أي التغذيات ستنتهي أولًا أو حتى ترتيبها، نظرًا لأنَّ الطلب غير محدَّد، فلا يمكننا الاعتماد على المنطق الذي

يقول أنَّ آخر تغذية ستبدأ هي التي ستنتهي أخيرًا.

هذا مثال على نمط ركام العمل (workpile pattern)، فيمكن تخيّل التغذيات المراد معالجتها كركام (pile) من المهام وخيط يمكنه أن يأخذ مهمة من هذا الركام، تمامًا كما لو كان لديك قائمة مهام وكل واحد تمثّل ملاحظة، فيمكنك اختيار الملاحظة العليا والقيام بكل ما ترغب به قبل الانتقال إلى الملاحظة الثانية، وهذه هي طريقة عمل هذا النمط.

سنصمّم مهامنا كدالة تسمى processFeed، والتي تقبل كائن Feed الذي يصف التغذية المراد معالجتها؛ إن تنفيذ هذه المهمة ليس مهمًا لهذا إليك هذا المثال:

```
fun processFeed(feed: Feed): Unit {
  println("Processing feed ${feed.name}")
}
```

سنفترض أننا أعطينا قائمة من التغذيات، ربما يمكننا قراءتها من قاعدة البيانات، سيرسل كل تغذية بدوره إلى مُنفِّذ Executor، وسيكون Executor الخاص بنا هو مجمّع الخيوط المؤقت (cached thread poo):

```
val executor = Executors.newCachedThreadPool()
```

وأخيرًا، سنحتاج إلى إرسال دالة عبر التنبيهات بمجرّد اكتمال جميع التغذيات:

```
fun sendNotification(): Unit {
  println("Sending notification")
}
```

حتى الآن، قمنا بتعدد الخيوط لمعالجة كل تغذية، لكن كيف يمكننا الآن التأكد من استدعاء الدالة sendNotification مرّة واحدة بمجرّد اكتمال جميع التغذيات؟ الفكرة الأولى هي استخدام عدّاد وتحديث التعداد بعد انتهاء كل مهمة تغذية، ومع ذلك، كيف ننتظر العداد؟ مرّة أخرى، يمكننا ببساطة هي زيادة العداد عند كل محاول لفتح القفل (spin lock) حتى نصل إلى العدد المطلوب.

الحل الأفضل هو تعطيل الخيط حتى يصبح جاهزًا، هذا هو مكان العد التنازلي للمزلاج، إذا أنشأنا مزلاجًا معيَّت إلى عدد التغذيات وعددنا تنازليًا كل المهام قبل أن تنتهي، فيمكننا بعد ذلك جعل الخيط الرئيسي ينتظر المزلاج،

وهذا هو المثال الكامل:

```
fun processFeed(feed: Feed): Unit {
   println("Processing feed ${feed.name}")
}

fun sendNotification(): Unit {
   println("Sending notification")
}

val feeds = listOf(
   Feed("Great Vegetable Store",
"http://www.greatvegstore.co.uk/items.xml"),
   Feed("Super Food Shop", "http://www.superfoodshop.com/products.csv")
)

val latch = CountDownLatch(feeds.size)

val executor = Executors.newCachedThreadPool()
for (feed in feeds) {
   executor.submit {
      processFeed(feed)
      latch.countDown()
   }
}

latch.await()
println("All feeds completed")
sendNotification()
```

سيُعطَّل الآن الخيط الرئيسي عند السطر `latch.await` ولن يستهلك المزيد من وقت CPU حتى يصبح جاهز للمضي قدمًا.

## خ. الحاجز الدوري

نوع أساسي متزامن آخر يشبه مزلاج العد التنازلي هو `CyclicBarrier`، والذي يسمح لخيوط متعدّدة بالانتظار حتى تصل جميعها إلى النقطة المطلوبة، الاستخدام الشائع للحاجز

(barrier) هو عندما يكون لديك مجموعة من الخيوط يجب أن تؤدي بعض المنطق ثم تنتظر حتى تصبح جميعها جاهزة قبل الانتقال.

لنتخيّل أنّنا نكتب نظامًا ينسخ ملفًا في أماكن متعدّدة، فنحن لا نرغب في بدء نسخ الملف الثاني قبل نسخ الملف الأول بنجاح في جميع الأماكن؛ كل مهمة تعمل في خيط منفصل والذي يكتب في مكان واحد، ويمكن تنفيذ هذه الحالة عن طريق تشغيل مهام متعدّدة على خيوط متعدّدة، وكل مهمة تهتم في مكان خرج واحد؛ يمكن لكل مهمة الانتظار على الحاجز ليبدأ الملف الثاني وذلك بمجرّد اكتمال النسخ في جميع المواضع.

لنعرّف أولًا مهمة تنسخ الملف بشكل متكرّر ثم تنتظر على الحاجز:

```
class CopyTask(val dir: Path, val paths: List<Path>, val barrier:
CyclicBarrier) : Runnable {

   override fun run() {
      for (path in paths) {
         val dest = dir.resolve(path)
         Files.copy(path, dest, StandardCopyOption.REPLACE_EXISTING)
         barrier.await()
      }
   }
}
```

بعد ذلك، أعدَّ منفِّذًا وأرسل المهام لكل موقع من مواقع الإخراج:

```
fun copyUsingBarrier(inputFiles: List<Path>, outputDirectories:
List<Path>) {
```

```
    val executor = Executors.newFixedThreadPool(outputDirectories.size)
    val barrier = CyclicBarrier(outputDirectories.size)

    for (dir in outputDirectories) {
        executor.submit {
            CopyTask(dir, inputFiles, barrier)
        }
    }
}
```

كما ترى، أحد المزايا وجود حاجز هو أنَّه يمكن إعادة استخدامه. في كل مرَّة يحرَّر فيها، فإنَّه على استعداد لاستخدامه مرَّة أخرى، يمكننا أيضًا استخدام مزلاج العد التنازلي هنا، لكنَّنا سنضطر إلى إنشاء واحد في كل مرة ومن ثم لدينا مشكلة مشاركة النسخة الجديد.

## د. الدخل والخرج الغير مُعطِّل والبرمجة غير المتزامنة

ركزنا خلال هذا الفصل على الخيوط كأداة التزامن الرئيسية، فهي حاسمة ومفيدة جدًا ولكن مع زيادة عدد الخيوط تقل الفائدة الهامشيَّة، فبوجود عدد أكبر من الخيوط، سيصرف المزيد من الوقت على تبديل السياق بينها؛ من الناحية المثالية، نرغب في أن نكون في الموقف حيث لدينا خيط واحد لكل نواة (core) لوحدة المعالجة المركزيّة، وتجنب سياق التبديل تمامًا، وهذا الهدف مستحيل إلى حد ما، ولكن يمكننا تقليل عدد المواضيع المستخدمة بشكل كبير.

تخيل مشكلة نريد فيها تنزيل عشرة تغذيات من موقع المورِّد، وبمجرَّد تنزيلها، نريد كتابتها إلى قاعدة البيانات الخاصة بنا، الحل لهذه المشكلة هو إنشاء عشرة خيوط يتلقى كل خيط منها تغذية واحدة.

نظرًا لأنَّ كل خيط ينتظر توفُّر المزيد من البيانات، فسيُعطَّل، وبتعطيل الخيوط أو انتهاء صلاحيَّة مدة المعالجة المُستقطعَة لها، سيقوم النظام بالتبديل بين الخيوط، ولو أردنا تكبير هذا النظام إلى 1000 تغذية، فسيكون هنالك تبديلات كثيرة عندما يُصرَف الجزء الأكبر من وقت المعالجة بالانتظار البيانات من الشبكة.

قد يكون الحل الأفضل هو أن يخطرنا نظام الدخل/الخرج I/O متى ما توافرت البيانات، ثمَّ يمكننا تخصيص

خيط لمعالجة هذه البيانات؛ ولإخطارنا، يجب علينا توفير دالة يشغلها نظام I/O عندما يكون جاهزًا، وهذه الدالـة أو الكتلـة يشـار إليهـا عـادة بـرد النـداء (callback)، وهـذه هي الفكـرة وراء I/O غـير المُعطّلـة. قـدّمت جافا نظـام دخل/خرج غير مُعطّل بدءًا من الإصدار 1.4 من JDK.

إذا استخدمنا نظام دخل/خرج غير مُعطّل لتنزيل جميع التغذيات من الموردين، فيجب أن نوفِّر عدة ردود نداء، ونظرًا لعدم وجود فكرة حول ترتيب التنفيذ، فسيحدّد هذا بالترتيب الذي ينتهي عنده التنزيل، وبعضها سيكون أكبر بكثـير من غيرهـا، ويشـار إلى هـذا النـوع من البرمجـة على أنَّه البرمجـة غـير المتزامنـة (asynchronous programming).

لا تعمل البرمجة غير المتزامنة على نظام الدخل/الخرج فقط، فقد يكون الحال أن لدينا رد نداء يعمل بمجرَّد أن ننهي عملية مقيّدة لوحدة المعالجة المركزية (CPU-bound operation) مثل حساب Pi لمئة ألف مكان ومن ثم تشغيل رد نداء لتكميل العملية.

رغم أنَّ هذه التقنيَّة قويّة جدًا، فيمكن أن تؤدي أيضا إلى ما يعرف باسم جحيم رد النداء (callback hell)، وهذا هو المكان الذي لدينا فيه مستويات متعدّدة من ردود النداء المتداخلة، حيث يؤدي كل رد نداء إلى المزيد من العمليات لتنفيذها:

```
fun persistOrder(order: Order, callback: (String) -> Unit): Unit = ...
fun chargeCard(card: Card, callback: (Boolean) -> Unit): Unit = ...
fun printInvoice(order: Order, callback: (Unit) -> Unit): Unit = ...

persistOrder(order, {
  println("Order has been saved; charging card")
  chargeCard(order.card, { result ->
  if (result) {
    println("Order has been charged; printing invoice")
    printInvoice(order, {
      println("Invoice has been printed")
    })
  }
})
```

```
})
```

كما ترى، تحتوي هذه الشيفرة البرمجيّة على ثلاثة مستويات من ردود النداء، في الحالة القصوى، ستكون المستويات بالعشرات، وعلى الرغم من أنّ هذا فعّال للغاية، لأنّ كل عملية ستعمل فقط بمجرّد اكتمال العملية السابقة ولن تحجب أي موارد أثناء الانتظار، فإنّها تؤدي إلى شيفرة برمجيّة غير قابلة للقراءة إلى حد ما.

## ذ. بُنَى المستقبل

تخيّل أنّنا نريد إرسال المهام إلى منفِّذ (executor)، لكنّنا نريد أن نعرف متى سيكتمل تنفيذها؛ إحدى الحلول هي تمرير متغير لكل مهمة، والتي يمكننا قراءة قيمته للتحقق من الحالة، ومع ذلك، فإن هذا يتطلّب منا إدارة تقلبات هذا المتغيّر، ويحتمل أن تحدث حالة قفل (spin locking) ريثما يجري التحقق منه.

الحل الأفضل سيكون بنيةً ما تمثّل حسابًا لم يكتمل بعد وستسمح هذه البنية بالحصول على قيمة الإرجاع فور اكتمالها، أو وضع عملية في الطابور لتعمل عليها عندما تكون جاهزة، أو تعطيلها حتى تنتهي. هذا النوع من البنى يسمى `future` (المستقبل)، يأتي الاسم من حقيقة أنَّه سيمثّل القيمة التي ستكون متوفّرة في وقت ما في المستقبل. (تسمى في بعض الأحيان الوعود `promises` في لغات أخرى مثل جافاسكربت. على الرغم في لغات أخرى مثل سكالا، إن بنى مثل `future` و `promise` هي بنيات مختلفة لكن ذات صلة.)

سنحتاج إلى دعم `ExecutorService` لإرجاع بنية `future` عند إرسال مهمة، ولفعل ذلك، سنحتاج إلى استخدام الواجهة `Callable` بدلًا من `Runnable`:

```
val executor = Executors.newFixedThreadPool(4)

val future: Future<Double> = executor.submit(Callable<Double> {
     Math.sqrt(15.64)
})
```

ترجع `future` البسيطة دوالًا للتحقق ما إذا انتهى وحصل على القيمة معطلًا استدعاء الخيط حتى يصبح جاهزًا.

إن القوّة الحقيقية، مع ذلك، تكمن في تجريد `CompletableFuture`، إذ تعزز هذه دعم `future` للعمليات

غير متزامنة وتعمل عن طريق ردود النداء بدلًا من تعطيل الخيط بشكل واضح. لإنشاء future مثل هذا، استخدام التوابع الثابتة المعرَّفة في الصنف، والتي تقبل اختياريًا، منفِّذًا:

```
val executor = Executors.newFixedThreadPool(4)
val future = CompletableFuture.supplyAsync(Supplier {  Math.sqrt(15.64) },
executor)
```

مع هذا future، يمكننا الآن إرفاق رد نداء:

```
future.thenApply {
    println("The square root has been calculated")
  }
```

يمكن سلسلة ردود النداء بحيث يمكن تمرير نتائج future إلى واحد آخر، فإذا زرنا مثال معالجة الطلب السابق، فيمكن كتابته على النحو التالي:

```
fun persistOrder(order: Order): String = TODO()
fun chargeCard(card: Card): Boolean = TODO()
fun printInvoice(order: Order): Unit = TODO()

CompletableFuture.supplyAsync {
   persistOrder(order)
}.thenApply { id ->
   println("Order has been saved; id is $id")
   chargeCard(order.card)
}.thenApply { result ->
if (result) {
   println("Order has been charged; printing invoice")
   printInvoice(order)
}
}
```

هذا أسهل قراءةً ويتجنّب العديد من المستويات المتداخلة من ردود النداء في حالة سَلسَلة ردود النداء. يمكن تنفيذ بنى future أيضًا معًا مع دمج النتائج في بنية future واحدة، فتخيَّل أنَّنا قرَّرنا الاستمرار في الطلب،

شحن البطاقة، وطباعة الفاتورة في وقت واحد مثل:

```
fun persistOrder(order: Order): CompletableFuture<String> = TODO()
fun chargeCard(card: Card): CompletableFuture<Boolean> = TODO()
fun printInvoice(order: Order): CompletableFuture<Unit> = TODO()

CompletableFuture.allOf(
   persistOrder(order),
   chargeCard(order.card),
   printInvoice(order)
).thenApply {
   println("Order is saved, charged and printed")
}
```

يملك `CompletableFuture` العديد من الدوال، مثل قبول أول قيمة مكتملة من عدة بنى `future`، وتعيين النتائج ومعالجة للأخطاء.

## 5. خلاصة الفصل

ركزنا في هذا الفصل على الأسس الأساسيَّة للتزامن في JVM، وكيفية استخدامها بفعالية في كوتلن. التزامن هو موضوع كبير، وقد وفَّر لك هذا الفصل قاعدة صلبة في هذا الموضوع إذا كنت جديدًا في هذا الموضوع، وبالنسبة لأولئك الذي يملكون معرفة جيِّدة بالتزامن وكتابة شيفرات متزامنة، قد رؤوا كيف تقدِّم كوتلن مساعدة صغيرة ودوال مساعدة مفيدة في الحزمة `concurrent` وصقلوا ما تعلموه.

## تفضل بزيارة مكتبة وادي التقنية للاطلاع على الكثير من الكتب التقنية المجانية





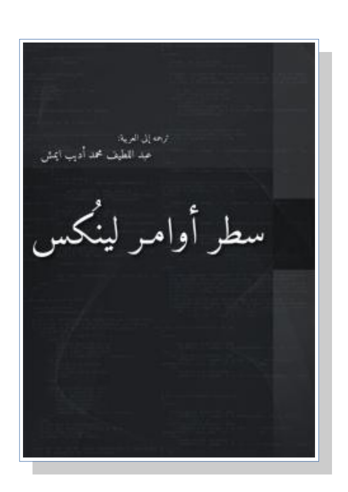














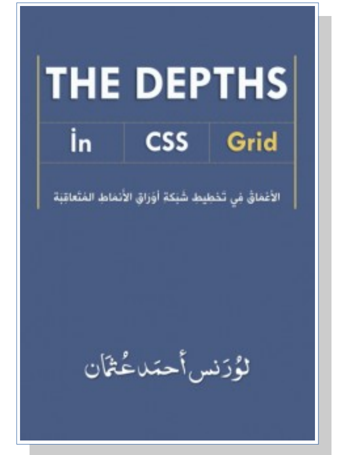


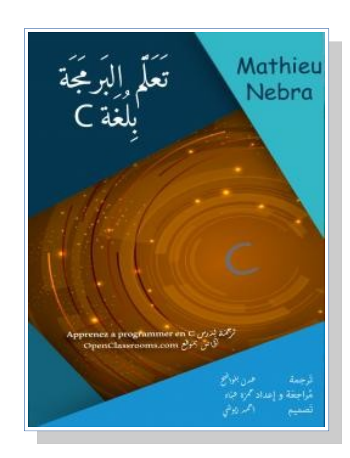




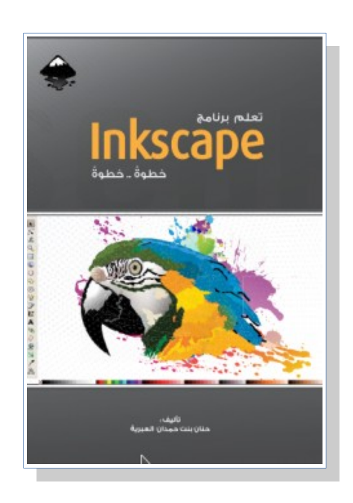
تعلم
استخدام JavaScript
للتعامل مع HTML
DOM
عبد اللطيف ايمش


القرية الهندسية
ENGINEERINGVILLAGE
احترف
الأردوينو
في 10 أيام
كتاب عملي ، تطبيقي ، مبسط


THE DEPTHS
in CSS Grid


تعلّم البرمجة
بلغة C
Mathieu
Nebra


php
تعلم البرمجة بلغة php
عبد اللطيف ايمش
أحمد ابو السعود
الطبعة الثانية


تعلم برنامج
Inkscape
خطوة ـ خطوة


وادي التقنية
تعلم لغة Go بسهولة
كل شيء تحتاج
لمعرفته
لتبدء البرمجة
مع لغة جو
بريان كيرنيغان
آلان دونوفان


القرية الهندسية
ENGINEERINGVILLAGE
احترف
سكراتش
في 10 أيام
SCRATCH


سلسلة تعلم ببساطة
راسبيري باي
ببساطة
عبدالله علي عبدالله


مدخل إلى:
علم تصميم البرمجيات
المبادرة العربية للترجمة العلمية


GNU
PSPP
http://www.gnu.org


خطوة
بخطوة
لنحترف
GIMP